

Oracle® Fusion Middleware

Developing Web User Interfaces with Oracle ADF Faces



12c (12.2.1.4.0)

E80056-01

September 2019

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Oracle Fusion Middleware Developing Web User Interfaces with Oracle ADF Faces, 12c (12.2.1.4.0)

E80056-01

Copyright © 2008, 2019, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Robin Whitmore (lead), Kathryn Munn, Cindy Hall, Walter Egan, Ralph Gordon, Vaibhav Gupta, Krithika Gangadhar, Sandhya Hombardi, Sandhya Sriram

Contributors: ADF Faces development team, Frank Nimphius, Laura Akel

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	xxxix
Documentation Accessibility	xxxix
Related Documents	xxxix
Conventions	xl

Part I Getting Started with ADF Faces

1 Introduction to ADF Faces

About ADF Faces	1-1
ADF Faces Framework	1-2
ADF Faces Components	1-4

2 ADF Faces Components Demo Application

About the ADF Faces Components Demo Application	2-1
Downloading and Installing the ADF Faces Components Demo Application	2-9

3 Getting Started with ADF Faces and JDeveloper

About Developing Declaratively in JDeveloper	3-1
Creating an Application Workspace	3-2
How to Create an ADF Faces Application Workspace	3-2
What Happens When You Create an Application Workspace	3-3
Defining Page Flows	3-4
How to Define a Page Flow	3-5
What Happens When You Use the Diagrammer to Create a Page Flow	3-6
Creating a View Page	3-7
How to Create JSF Pages	3-10
What Happens When You Create a JSF Page	3-10
What You May Need to Know About Updating Your Application to Use the Facelets Engine	3-15

What You May Need to Know About Automatic Component Binding	3-16
How to Add ADF Faces Components to JSF Pages	3-20
What Happens When You Add Components to a Page	3-23
How to Set Component Attributes	3-24
What Happens When You Use the Properties window	3-26
Creating EL Expressions	3-26
How to Create an EL Expression	3-27
How to Use the EL Format Tags	3-29
How to Use EL Expressions Within Managed Beans	3-30
Creating and Using Managed Beans	3-31
How to Create a Managed Bean in JDeveloper	3-31
What Happens When You Use JDeveloper to Create a Managed Bean	3-33
What You May Need to Know About Component Bindings and Managed Beans	3-34
Viewing ADF Faces Javadoc	3-35
How to View ADF Faces Source Code and Javadoc	3-35

Part II Understanding ADF Faces Architecture

4 Using ADF Faces Client-Side Architecture

About Using ADF Faces Architecture	4-1
Adding JavaScript to a Page	4-3
How to Use Inline JavaScript	4-3
How to Import JavaScript Libraries	4-4
What You May Need to Know About Accessing Client Event Sources	4-5
Instantiating Client-Side Components	4-5
How to Configure a Component to for a Client-Side Instance	4-5
What Happens When You Set clientComponent to true	4-6
Listening for Client Events	4-7
How to Listen for Client Events	4-7
How to Use an ADF Client Listener to Control Navigating Away From a JSF Page	4-8
Accessing Component Properties on the Client	4-9
Secure Client Properties	4-10
How to Set Property Values on the Client	4-13
What Happens at Runtime: How Client Properties Are Set on the Client	4-14
How to Unsecure the disabled Property	4-14
How to "Unsynchronize" Client and Server Property Values	4-15
Using Bonus Attributes for Client-Side Components	4-18
How to Create Bonus Attributes	4-18
What You May Need to Know About Marshalling Bonus Attributes	4-19

Understanding Rendering and Visibility	4-19
How to Set Visibility Using JavaScript	4-20
What You May Need to Know About Visible and the isShowing Function	4-22
Locating a Client Component on a Page	4-22
What You May Need to Know About Finding Components in Naming Containers	4-23
JavaScript Library Partitioning	4-24
How to Create JavaScript Partitions	4-25
What Happens at Runtime: JavaScript Partitioning	4-26

5 Using the JSF Lifecycle with ADF Faces

About Using the JSF Lifecycle and ADF Faces	5-1
Using the Immediate Attribute	5-4
How to Use the Immediate Attribute	5-8
Using the Optimized Lifecycle	5-8
Using the Client-Side Lifecycle	5-9
Using Subforms to Create Sections on a Page	5-10
Object Scope Lifecycles	5-11
Passing Values Between Pages	5-13
How to Use the pageFlowScope Scope Within Java Code	5-14
How to Use the pageFlowScope Scope Without Writing Java Code	5-15
What Happens at Runtime: How Values Are Passed	5-16

6 Handling Events

About Events and Event Handling	6-1
Events and Partial Page Rendering	6-2
Event and Even Root Components	6-3
Client-Side Event Model	6-4
Using ADF Faces Server Events	6-4
How to Handle Server-Side Events	6-6
Using JavaScript for ADF Faces Client Events	6-7
ADF Faces Client-Side Events	6-9
How to Use Client-Side Events	6-11
How to Return the Original Source of the Event	6-13
How to Use Client-Side Attributes for an Event	6-14
How to Block UI Input During Event Execution	6-15
How to Prevent Events from Propagating to the Server	6-15
How to Indicate No Response is Expected	6-16
What Happens at Runtime: How Client-Side Events Work	6-16
What You May Need to Know About Using Naming Containers	6-17

Sending Custom Events from the Client to the Server	6-17
How to Send Custom Events from the Client to the Server	6-18
What Happens at Runtime: How Client and Server Listeners Work Together	6-20
What You May Need to Know About Marshalling and Unmarshalling Data	6-20
Mapping Java to JavaScript	6-21
Executing a Script Within an Event Response	6-21
Using ADF Faces Client Behavior Tags	6-23
How to Use the scrollComponentIntoViewBehavior Tag	6-24
Using Polling Events to Update Pages	6-25
How to Use the Poll Component	6-26

7 Validating and Converting Input

About ADF Faces Converters and Validators	7-1
ADF Faces Converters and Validators Use Cases and Examples	7-2
Additional Functionality for ADF Faces Converters and Validators	7-2
Conversion, Validation, and the JSF Lifecycle	7-2
Adding Conversion	7-3
How to Add a Converter	7-4
How to Specify Multiple Converter Patterns	7-5
How to Specify Negative Numbers for Converters	7-6
What Happens at Runtime: How Converters Work	7-6
What You May Need to Know About Number Converters	7-6
What You May Need to Know About Date Time Converters	7-7
Creating Custom ADF Faces Converters	7-9
How to Create a Custom ADF Faces Converter	7-9
Implement Server-Side (Java) Conversion	7-9
Register ADF Faces Converter in faces-config.xml	7-10
Create a Client-Side Version of the Converter	7-10
Modify the Server Converter to Enable Client Conversion	7-11
Using Custom ADF Faces Converter on a JSF Page	7-14
What You May Need to Know About Custom ADF Faces Converters	7-14
How to Declaratively Register a Client-Side Only Converter Script	7-15
Adding Validation	7-17
How to Add Validation	7-18
Using Validation Attributes	7-18
Using ADF Faces Validators	7-18
What Happens at Runtime: How Validators Work	7-20
What You May Need to Know About Multiple Validators	7-21
Creating Custom JSF Validation	7-21
How to Create a Backing Bean Validation Method	7-21

What Happens When You Create a Backing Bean Validation Method	7-22
How to Create a Custom JSF Validator	7-22
What Happens When You Use a Custom JSF Validator	7-24

8 Rerendering Partial Page Content

About Partial Page Rendering	8-1
Using Partial Triggers	8-4
How to Use Partial Triggers	8-6
What You May Need to Know About Using the Browser Back Button	8-8
What You May Need to Know About PPR and Screen Readers	8-9
Using the Target Tag to Execute PPR	8-9
How to Use the Target Tag to Enable Partial Page Rendering	8-10
Enabling Partial Page Rendering Programmatically	8-13
How to Enable Partial Page Rendering Programmatically	8-13
Using Partial Page Navigation	8-14
How to Use Partial Page Navigation	8-15
What You May Need to Know About PPR Navigation	8-15
Using a Streaming Component to Allow Page Loading	8-16
How to Implement an Instance of AsyncFetch to Fetch Streaming Data	8-16
How to Create a Streaming Component	8-17

Part III Creating Your Layout

9 Organizing Content on Web Pages

About Organizing Content on Web Pages	9-1
ADF Faces Layout Components	9-2
Additional Functionality for Layout Components	9-6
Starting to Lay Out a Page	9-6
Geometry Management and Component Stretching	9-7
Nesting Components Inside Components That Allow Stretching	9-9
Using Quick Start Layouts	9-12
Tips for Using Geometry-Managed Components	9-13
How to Configure the document Tag	9-14
Arranging Content in a Grid	9-17
How to Use the panelGridLayout, gridRow, and gridColumn Components to Create a Grid-Based Layout	9-20
What You May Need to Know About Geometry Management and the panelGridLayout Component	9-24
What You May Need to Know About Determining the Structure of Your Grid	9-25

What You May Need to Know About Determining Which Layout Component to Use	9-27
Displaying Contents in a Dynamic Grid Using a masonryLayout Component	9-28
How to Use a masonryLayout Component	9-31
Achieving Responsive Behavior Using matchMediaBehavior Tag	9-33
Arranging Contents to Stretch Across a Page	9-38
How to Use the panelStretchLayout Component	9-39
What You May Need to Know About Geometry Management and the panelStretchLayout Component	9-42
Using Splitters to Create Resizable Panes	9-43
How to Use the panelSplitter Component	9-45
What You May Need to Know About Geometry Management and the panelSplitter Component	9-49
Arranging Page Contents in Predefined Fixed Areas	9-50
How to Use the panelBorderLayout Component to Arrange Page Contents in Predefined Fixed Areas	9-52
Arranging Content in Forms	9-53
How to Use the panelFormLayout Component	9-55
What You May Need to Know About Responsive Mode in the panelFormLayout Component	9-59
What You May Need to Know About Using the group Component with the panelFormLayout Component	9-61
Arranging Contents in a Dashboard	9-64
How to Use the panelDashboard Component	9-68
What You May Need to Know About Geometry Management and the panelDashboard Component	9-71
Displaying and Hiding Contents Dynamically	9-72
How to Use the showDetail Component	9-78
How to Use the showDetailHeader Component	9-80
How to Use the panelBox Component	9-84
What You May Need to Know About Disclosure Events	9-86
What You May Need to Know About Skinning and the showDetail Component	9-87
What You May Need to Know About Skinning and the showDetailHeader Component	9-88
What You May Need to Know About Skinning and the panelBox Component	9-88
Displaying or Hiding Contents in Panels	9-88
How to Use the panelAccordion Component	9-96
How to Use the panelTabbed Component	9-99
How to Use the panelDrawer Component	9-102
How to Use the panelSpringboard Component	9-104
What You May Need to Know About Switching Between Grid and Strip Mode	9-105
How to Use the showDetailItem Component to Display Content	9-105

What You May Need to Know About Geometry Management and the showDetailItem Component	9-110
What You May Need to Know About showDetailItem Disclosure Events	9-112
What You May Need to Know About Skinning and the panelTabbed Component	9-113
Adding a Transition Between Components	9-116
How to Use the Deck Component	9-117
What You May Need to Know About Geometry Management and the deck Component	9-119
Displaying Items in a Static Box	9-120
How to Use the panelHeader Component	9-122
How to Use the decorativeBox Component	9-126
What You May Need to Know About Geometry Management and the decorativeBox Component	9-127
What You May Need to Know About Skinning and the panelHeader Component	9-128
What You May Need to Know About Skinning and the decorativeBox Component	9-128
Displaying a Bulleted List in One or More Columns	9-129
How to Use the panelList Component	9-130
What You May Need to Know About Creating a List Hierarchy	9-131
Grouping Related Items	9-132
How to Use the panelGroupLayout Component	9-135
What You May Need to Know About Geometry Management and the panelGroupLayout Component	9-137
Separating Content Using Blank Space or Lines	9-137
How to Use the spacer Component	9-139
How to Use the Separator Component	9-139

10 Creating and Reusing Fragments, Page Templates, and Components

About Reusable Content	10-1
Reusable Components Use Cases and Examples	10-3
Additional Functionality for Reusable Components	10-4
Using Page Templates	10-4
How to Create a Page Template	10-9
What Happens When You Create a Page Template	10-14
How to Create JSF Pages Based on Page Templates	10-14
What Happens When You Use a Template to Create a Page	10-17
What Happens at Runtime: How Page Templates Are Resolved	10-17
What You May Need to Know About Page Templates and Naming Containers	10-18
Using Page Fragments	10-18
How to Create a Page Fragment	10-21

What Happens When You Create a Page Fragment	10-22
How to Use a Page Fragment in a JSF Page	10-23
What Happens at Runtime: How Page Fragments are Resolved	10-23
What You May Need to Know About Accessing a Page From a Different View ID	10-23
Using Declarative Components	10-24
How to Create a Declarative Component	10-27
What Happens When You Create a Declarative Component	10-32
How to Deploy Declarative Components	10-34
How to Use Declarative Components in JSF Pages	10-34
What Happens When You Use a Declarative Component on a JSF Page	10-36
What Happens at Runtime: Declarative Components	10-37
Adding Resources to Pages	10-37
How to Add Resources to Page Templates and Declarative Components	10-38
What Happens at Runtime: How to Add Resources to the Document Header	10-38

Part IV Using Common ADF Faces Components

11 Using Input Components and Defining Forms

About Input Components and Forms	11-1
Input Component Use Cases and Examples	11-3
Additional Functionality for Input Components and Forms	11-5
Defining Forms	11-6
How to Add a Form to a Page	11-8
How to Add a Subform to a Page	11-8
How to Add a Button to Reset the Form	11-9
Using the inputText Component	11-9
How to Add an inputText Component	11-11
How to Add the Ability to Insert Text into an inputText Component	11-14
Using the Input Number Components	11-15
How to Add an inputNumberSlider or an inputRangeSlider Component	11-17
How to Add an inputNumberSpinbox Component	11-18
Using Color and Date Choosers	11-18
How to Add an inputColor Component	11-20
How to Add an InputDate Component	11-22
What You May Need to Know About Including a Default Value for an InputDate Component	11-25
What You May Need to Know About Setting the Time Value for an InputDate Component	11-26
What You May Need to Know About Selecting Time Zones Without the inputDate Component	11-26

What You May Need to Know About Multi-Selection Support in the chooseDate Component	11-28
What You May Need to Know About Creating a Custom Time Zone List	11-30
Using Selection Components	11-31
How to Use Selection Components	11-35
What You May Need to Know About the contentDelivery Attribute on the SelectManyChoice Component	11-38
Using Shuttle Components	11-39
How to Add a selectManyShuttle or selectOrderShuttle Component	11-41
What You May Need to Know About Using a Client Listener for Selection Events	11-43
Using the richTextEditor Component	11-44
How to Add a richTextEditor Component	11-47
How to Add the Ability to Insert Text into a richTextEditor Component	11-49
How to Customize the Toolbar	11-50
About richTextEditor for UIWebView User-Agent	11-52
Using File Upload	11-52
How to Use the inputFile Component	11-56
How to Configure the inputFile Component to Upload Multiple Files	11-57
What You May Need to Know About Temporary File Storage	11-59
What You May Need to Know About Uploading Multiple Files	11-60
What You May Need to Know About Customizing User Interface of inputFile Component	11-61
Using Code Editor	11-63
How to Add a codeEditor Component	11-67

12 Using Tables, Trees, and Other Collection-Based Components

About Collection-Based Components	12-1
Collection-Based Component Use Cases and Examples	12-3
Additional Functionality for Collection-Based Components	12-6
Common Functionality in Collection-Based Components	12-7
Displaying Data in Rows and Nodes	12-7
Content Delivery	12-8
Row Selection	12-13
Editing Data in Tables, Trees, and Tree Tables	12-14
Using Popup Dialogs in Tables, Trees, and Tree Tables	12-16
Accessing Client Collection Components	12-18
Geometry Management for the Table, Tree, and Tree Table Components	12-18
Displaying Data in Tables	12-21
Columns and Column Data	12-22
Formatting Tables	12-23

Formatting Columns	12-25
How to Display a Table on a Page	12-28
What Happens When You Add a Table to a Page	12-41
What Happens at Runtime: Data Delivery	12-42
What You May Need to Know About Programmatically Enabling Sorting for Table Columns	12-42
What You May Need to Know About Performing an Action on Selected Rows in Tables	12-43
What You May Need to Know About Dynamically Determining Values for Selection Components in Tables	12-44
What You May Need to Know About Read Only Tables	12-45
Adding Hidden Capabilities to a Table	12-46
How to Use the detailStamp Facet	12-48
What Happens at Runtime: The rowDisclosureEvent	12-49
Enabling Filtering in Tables	12-50
How to Add Filtering to a Table	12-51
Displaying Data in Trees	12-52
How to Display Data in Trees	12-55
What Happens When You Add a Tree to a Page	12-58
What Happens at Runtime: Tree Component Events	12-59
What You May Need to Know About Programmatically Expanding and Collapsing Nodes	12-59
What You May Need to Know About Programmatically Selecting Nodes	12-61
Displaying Data in Tree Tables	12-61
How to Display Data in a Tree Table	12-63
Passing a Row as a Value	12-63
Displaying Table Menus, Toolbars, and Status Bars	12-64
How to Add a panelCollection with a Table, Tree, or Tree Table	12-66
Displaying a Collection in a List	12-68
How to Display a Collection in a List	12-72
What You May Need to Know About Scrollbars in a List View	12-73
Displaying Images in a Carousel	12-74
How to Create a Carousel	12-79
What You May Need to Know About the Carousel Component and Different Browsers	12-85
Exporting Data from Table, Tree, or Tree Tables	12-85
How to Export Table, Tree, or Tree Table Data to an External Format	12-88
What Happens at Runtime: How Row Selection Affects the Exported Data	12-90
Accessing Selected Values on the Client from Collection-Based Components	12-90
How to Access Values from a Selection in Stamped Components.	12-90
What You May Need to Know About Accessing Selected Values	12-92

13 Using List-of-Values Components

About List-of-Values Components	13-1
Additional Functionality for List-of-Values Components	13-9
Creating the ListOfValues Data Model	13-10
How to Create the ListOfValues Data Model	13-10
Using the inputListOfValues Component	13-11
How to Use the InputListOfValues Component	13-11
What You May Need to Know About Dynamically Creating Auto Suggest Behavior for LOV Components	13-14
What You May Need to Know About Skinning the Search and Select Dialogs in the LOV Components	13-14
Using the InputComboboxListOfValues Component	13-15
How to Use the InputComboboxListOfValues Component	13-15
Using the InputSearch Component	13-17
How to Use the InputSearch Component	13-21
What You May Need to Know About InputSearch Component Tags and REST Data	13-22
What You May Need to Know About InputSearch Attributes	13-23
What You May Need to Know About SearchSection Attributes	13-33
How to Customize InputSearch Component Display Modes	13-36
How to Add Custom Buttons to Suggestions Popup	13-39
What You May Need to Know About SuggestionSection Component	13-41
How to Set Attributes of Suggestion Section	13-41
What You May Need to Know About suggestionSection Attributes	13-42
How to Specify Dependency-Based Filtering on InputSearch Components	13-44

14 Using Query Components

About Query Components	14-1
Query Component Use Cases and Examples	14-4
Additional Functionality for the Query Components	14-4
Creating the Query Data Model	14-4
How to Create the Query Data Model	14-15
Using the quickQuery Component	14-16
How to Add the quickQuery Component Using a Model	14-16
How to Use a quickQuery Component Without a Model	14-17
What Happens at Runtime: How the Framework Renders the quickQuery Component and Executes the Search	14-19
Using the query Component	14-19
How to Add the Query Component	14-24

15 Using Menus, Toolbars, and Toolboxes

About Menus, Toolbars, and Toolboxes	15-1
Menu Components Use Cases and Examples	15-2
Additional Functionality for Menu and Toolbar Components	15-4
Using Menus in a Menu Bar	15-5
How to Create and Use Menus in a Menu Bar	15-10
Using Toolbars	15-17
How to Create and Use Toolbars	15-19
What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined	15-24
What You May Need to Know About Toolbars	15-24

16 Using Popup Dialogs, Menus, and Windows

About Popup Dialogs, Menus, and Windows	16-1
Popup Dialogs, Menus, Windows Use Cases and Examples	16-3
Additional Functionality for Popup Dialogs, Menus, and Windows	16-4
What You May Need to Know About ADF Faces Window Manager Configuration	16-4
Declaratively Creating Popups	16-6
How to Create a Dialog	16-8
How to Create a Panel Window	16-13
How to Create a Context Menu	16-16
How to Create a Note Window	16-17
What Happens at Runtime: Popup Component Events	16-19
What You May Need to Know About Dialog Events	16-20
What You May Need to Know About Animation and Popups	16-21
Controlling Display Behavior of Popups	16-22
How to Dismiss a Popup Component Automatically	16-23
What Happens When a Popup Component is Automatically Dismissed	16-23
Declaratively Invoking a Popup	16-23
How to Declaratively Invoke a Popup Using the af:showPopupBehavior Tag	16-24
What Happens When You Use af:showPopupBehavior Tag to Invoke a Popup	16-25
Programmatically Invoking a Popup	16-26
How to Programmatically Invoke a Popup	16-28
What Happens When You Programmatically Invoke a Popup	16-28
Displaying Contextual Information in Popups	16-29
How to Create Contextual Information	16-30
Controlling the Automatic Cancellation of Inline Popups	16-31
How to Disable the Automatic Cancellation of an Inline Popup	16-32

What Happens When You Disable the Automatic Cancellation of an Inline Popup	16-32
Resetting Input Fields in a Popup	16-33
How to Reset the Input Fields in a Popup	16-34
What Happens When You Configure a Popup to Reset Its Input Fields	16-34

17 Using a Calendar Component

About Creating a Calendar Component	17-1
Calendar Use Cases and Examples	17-4
Additional Functionality for the Calendar	17-5
Creating the Calendar	17-6
Calendar Classes	17-6
How to Create a Calendar	17-7
Configuring the Calendar Component	17-8
How to Configure the Calendar Component	17-8
What Happens at Runtime: Calendar Events and PPR	17-12
Adding Functionality Using Popup Components	17-13
How to Add Functionality Using Popup Components	17-14
Customizing the Toolbar	17-16
How to Customize the Toolbar	17-17
Styling the Calendar	17-19
How to Style Activities	17-20
What Happens at Runtime: Activity Styling	17-22
How to Customize Dates	17-22

18 Using Output Components

About Output Text, Image, Icon, and Media Components	18-1
Output Components Use Case and Examples	18-2
Additional Functionality for Output Components	18-3
Displaying Output Text and Formatted Output Text	18-3
How to Display Output Text	18-5
What You May Need to Know About Allowed Format and Character Codes in the outputFormatted Component	18-6
Displaying Icons	18-8
How to Display Icons	18-8
Displaying Images	18-9
How to Display Images	18-9
Using Images as Links	18-10
How to Use Images as Links	18-10
Displaying Application Status Using Icons	18-11

Playing Video and Audio Clips	18-12
How to Allow Playing of Audio and Video Clips	18-13

19 Displaying Tips, Messages, and Help

About Displaying Tips and Messages	19-1
Messaging Components Use Cases and Examples	19-4
Additional Functionality for Message Components	19-8
Displaying Tips for Components	19-9
How to Display Tips for Components	19-9
Displaying Hints and Error Messages for Validation and Conversion	19-10
How to Define Custom Validator and Converter Messages for a Component Instance	19-13
How to Define Custom Validator and Converter Messages for All Instances of a Component	19-14
How to Display Component Messages Inline	19-15
How to Display Global Messages Inline	19-16
What Happens at Runtime: How Messages Are Displayed	19-16
Grouping Components with a Single Label and Message	19-17
How to Group Components with a Single Label and Message	19-19
Displaying Help for Components	19-20
How to Create Help Providers	19-24
How to Create a Resource Bundle-Based Provider	19-25
How to Create an XLIFF Provider	19-26
How to Create a Managed Bean Provider	19-27
How to Create an External URL Help Provider	19-28
How to Create a Custom Java Class Help Provider	19-30
How to Register the Help Provider	19-31
How to Access Help Content from a UI Component	19-34
How to Use JavaScript to Launch an External Help Window	19-34
What You May Need to Know About Skinning and Definition Help	19-35
Combining Different Message Types	19-36

20 Working with Navigation Components

About Navigation Components	20-1
Navigation Components Use Cases and Examples	20-2
Additional Functionality for Navigation Components	20-6
Common Functionality in Navigation Components	20-7
Using Buttons and Links for Navigation	20-7
How to Use Buttons and Links for Navigation and Deliver ActionEvents	20-9
How to Use Buttons and Links for Navigation Without Delivering ActionEvents	20-12

What You May Need to Know About Using Partial Page Navigation	20-13
Configuring a Browser's Context Menu for Links	20-13
How to Configure a Browser's Context Menu for Command Links	20-14
What Happens When You Configure a Browser's Context Menu for Command Links	20-15
Using Buttons or Links to Invoke Functionality	20-15
How to Use an Action Component to Download Files	20-15
How to Use an Action Component to Reset Input Fields	20-18
Using Navigation Items for a Page Hierarchy	20-19
How to Create Navigation Cases for a Page Hierarchy	20-23
Using a Menu Model to Create a Page Hierarchy	20-24
How to Create the Menu Model Metadata	20-26
What Happens When You Use the Create ADF Menu Model Wizard	20-34
How to Bind the navigationPane Component to the Menu Model	20-35
How to Use the breadCrumbs Component with a Menu Model	20-39
How to Use the menuBar Component with a Menu Model	20-41
What Happens at Runtime: How the Menu Model Creates a Page Hierarchy	20-43
What You May Need to Know About Using Custom Attributes	20-45
Creating a Simple Navigational Hierarchy	20-46
How to Create a Simple Page Hierarchy	20-48
How to Use the breadCrumbs Component	20-52
What You May Need to Know About Removing Navigation Tabs	20-53
What You May Need to Know About the Size of Navigation Tabs	20-54
What You May Need to Know About Skinning and Navigation Tabs	20-54
Using Train Components to Create Navigation Items for a Multistep Process	20-55
How to Create the Train Model	20-60
How to Configure Managed Beans for the Train Model	20-62
How to Bind to the Train Model in JSF Pages	20-66

21 Determining Components at Runtime

About Determining Components at Runtime	21-1
Creating the Model for a Dynamic Component	21-4
How to Create the Model Without Groups	21-4
How to Create the Model Using Groups	21-7
Adding a Dynamic Component as a Form to a Page	21-9
How to Add a Dynamic Component as a Form without Groups to a Page	21-9
How to Add a Dynamic Component as a Form with Groups to a Page	21-10
Adding a Dynamic Component as a Table to a Page	21-12
How to Add a Dynamic Component as a Table Without Groups to a Page	21-13
How to Add a Dynamic Component as a Table with Groups to a Page	21-13
Using Validation and Conversion with Dynamic Components	21-15

Part V Using ADF Data Visualization Components

22 Introduction to ADF Data Visualization Components

About ADF Data Visualization Components	22-1
Chart Component Use Cases and Examples	22-1
Picto Chart Use Cases and Examples	22-3
Gauge Component Use Cases and Examples	22-6
NBox Use Cases and Examples	22-8
Pivot Table Component Use Cases and Examples	22-11
Geographic Map Component Use Cases and Examples	22-11
Thematic Map Component Use Cases and Examples	22-12
Gantt Chart Component Use Cases and Examples	22-13
Timeline Component Use Cases and Examples	22-14
Hierarchy Viewer Component Use Cases and Examples	22-15
Treemap and Sunburst Components Use Cases and Examples	22-16
Diagram Use Cases and Examples	22-18
Tag Cloud Component Use Cases and Examples	22-21
Additional Functionality for Data Visualization Components	22-22
Common Functionality in Data Visualization Components	22-23
Content Delivery	22-23
Automatic Partial Page Rendering (PPR)	22-25
Active Data Support	22-25
Text Resources from Application Resource Bundles	22-26
Providing Data for ADF Data Visualization Components	22-27

23 Using Chart Components

About the Chart Component	23-1
Chart Component Use Cases and Examples	23-1
End User and Presentation Features of Charts	23-9
Chart Data Labels	23-9
Chart Element Labels	23-10
Chart Sizing	23-10
Chart Legends	23-11
Chart Styling	23-11
Chart Series Hiding	23-12
Chart Reference Objects	23-12
Chart Series Effects	23-13

Chart Series Customization	23-13
Chart Data Cursor	23-14
Chart Time Axis	23-15
Chart Categorical Axis	23-16
Chart Popups and Context Menus	23-18
Chart Selection Support	23-18
Chart Zoom and Scroll	23-19
Legend and Marker Dimming	23-20
Pie Chart Other Slice Support	23-21
Exploding Slices in Pie Charts	23-22
Active Data Support (ADS)	23-22
Chart Animation	23-23
Chart Image Formats	23-23
Additional Functionality for Chart Components	23-23
Using the Chart Component	23-24
Chart Component Data Requirements	23-24
Area, Bar, and Line Chart Data Requirements	23-25
Bubble Chart Data Requirements	23-25
Combination Chart Data Requirements	23-25
Funnel Chart Data Requirements	23-26
Pie Chart Data Requirements	23-26
Scatter Chart Data Requirements	23-26
Spark Chart Data Requirements	23-26
Stock Chart Data Requirements	23-26
Configuring Charts	23-27
How to Add a Chart to a Page	23-31
What Happens When You Add a Chart to a Page	23-34
Adding Data to Charts	23-34
How to Add Data to Area, Bar, Combination, and Line Charts	23-34
How to Add Data to Pie Charts	23-38
How to Add Data to Bubble or Scatter Charts	23-41
How to Add Data to Funnel Charts	23-44
How to Add Data to Stock Charts	23-46
How to Add Data to Spark Charts	23-49
Customizing Chart Display Elements	23-53
How to Configure Chart Labels	23-53
How to Configure Chart Data Labels	23-54
How to Configure Chart Element Labels	23-56
How to Configure Chart Axis Labels	23-57
How to Configure Chart Legends	23-58
How to Format Chart Numerical Values	23-60

How to Format Numeric Values on a Chart's Value, Value Label, or Axis Values	23-62
How to Format Numeric Values on a Chart's Axis Label	23-62
How to Specify Numeric Patterns, Currency, or Percent	23-63
Customizing a Chart Axis	23-63
How to Configure a Time Axis	23-63
How to Customize the Chart Axis	23-67
Configuring Dual Y-Axis	23-71
Adding Reference Objects to a Chart	23-71
How to Add a Reference Object to a Chart	23-75
What You May Need to Know About Adding Reference Objects to Charts	23-77
How to Configure a Stacked Chart	23-77
Customizing Chart Series	23-78
How to Customize a Chart Series	23-79
How to Configure Series Fill Effects on All Series in a Chart	23-82
Customizing Chart Groups	23-83
How to Customize a Chart Group	23-83
How to Configure Hierarchical Labels Using Chart Groups	23-84
How to Configure the Pie Chart Other Slice	23-87
How to Explode Pie Chart Slices	23-88
How to Configure Animation	23-89
What You May Need to Know About Skinning and Customizing Chart Display Elements	23-90
Adding Interactive Features to Charts	23-90
How to Add a Data Cursor	23-90
How to Configure Hide and Show Behavior	23-91
How to Configure Legend and Marker Dimming	23-93
How to Configure Selection Support	23-93
How to Configure Popups and Context Menus	23-96
How to Configure Chart Zoom and Scroll	23-97
How to Configure Chart Overview Window	23-99

24 Using Picto Chart Components

About the Picto Chart Component	24-1
Picto Chart Use Cases and Examples	24-1
End User and Presentation Features of Picto Charts	24-4
Additional Functionality for Picto Chart Components	24-5
Using the Picto Chart Component	24-6
Picto Chart Data Requirements	24-6
How to Add a Picto Chart to a Page	24-7
What Happens When You Add a Picto Chart to a Page	24-8

25 Using Gauge Components

About the Gauge Component	25-1
Gauge Component Use Cases and Examples	25-1
End User and Presentation Features of Gauge Components	25-3
Gauge Shape Variations	25-4
Gauge Thresholds	25-5
Gauge Visual Effects	25-5
Active Data Support (ADS)	25-6
Gauge Animation	25-6
Gauge Tooltips	25-6
Gauge Popups and Context Menus	25-7
Gauge Value Change Support	25-7
Gauge Reference Lines (Status Meter Gauges)	25-8
Additional Functionality of Gauge Components	25-8
Using the Gauge Component	25-9
Gauge Component Data Requirements	25-9
How to Add a Gauge to a Page	25-10
What Happens When You Add a Gauge to a Page	25-12
How to Add Data to Gauges	25-12
Configuring Gauges	25-14
Configuring Dial Gauges	25-15
Configuring LED Gauges	25-15
Configuring Rating Gauges	25-15
Configuring Status Meter Gauges	25-15
Customizing Gauge Display Elements	25-16
How to Configure Gauge Thresholds	25-16
Formatting Gauge Style Elements	25-19
How to Change Gauge Size and Apply CSS Styles	25-19
How to Format Gauge Text	25-20
What You May Need to Know About Skinning and Formatting Gauge Style Elements	25-21
How to Format Numeric Data Values in Gauges	25-22
How to Disable Gauge Visual Effects	25-23
How to Configure Gauge Animation	25-23
How to Configure Status Meter Gauge Reference Lines	25-24
Adding Interactivity to Gauges	25-25
How to Configure Gauge Tooltips	25-25
How to Add a Popup or Context Menu to a Gauge	25-26

26 Using NBox Components

About the NBox Component	26-1
NBox Use Cases and Examples	26-1
End User and Presentation Features of NBoxes	26-4
Additional Functionality for NBox Components	26-4
Using the NBox Component	26-5
NBox Data Requirements	26-5
Configuring NBoxes	26-6
How to Add an NBox to a Page	26-7
What Happens When You Add an NBox to a Page	26-9

27 Using Pivot Table Components

About the Pivot Table Component	27-1
Pivot Table and Pivot Filter Bar Component Use Cases and Examples	27-1
End User and Presentation Features of Pivot Table Components	27-4
Pivot Filter Bar	27-4
Pivoting	27-4
Editing Data Cells	27-5
Data and Header Sorting	27-7
Drilling	27-8
Scrolling and Page Controls	27-9
Persistent Header Layers	27-11
Split View of Large Data Sets	27-11
Sizing	27-13
Header Cell Word Wrapping	27-14
Active Data Support (ADS)	27-15
Additional Functionality for the Pivot Table Component	27-15
Using the Pivot Table Component	27-16
Pivot Table Data Requirements	27-17
Configuring Pivot Tables	27-17
How to Add a Pivot Table to a Page	27-18
Configuring Pivot Table Display Size and Style	27-21
What Happens When You Add a Pivot Table to a Page	27-22
What You May Need to Know About Displaying Large Data Sets	27-22
What You May Need to Know About Pivot Tables on Touch Devices	27-23
What You May Need to Know About Skinning and Customizing the Appearance of Pivot Tables	27-23
Configuring Header and Data Cell Stamps	27-23

Using var and varStatus Properties	27-24
How to Configure Header and Data Cell Stamps	27-27
Using Pivot Filter Bars	27-31
Using a Pivot Filter Bar with a Pivot Table	27-32
Using a Pivot Filter Bar with a Graph	27-32
What You May Need to Know About Skinning and Customizing the Appearance of Pivot Filter Bars	27-33
Adding Interactivity to Pivot Tables	27-33
Using Selection in Pivot Tables	27-33
Using Partial Page Rendering	27-34
Exporting from a Pivot Table	27-34
Displaying Pivot Tables in Printable Pages	27-36
Formatting Pivot Table Cell Content With CellFormat	27-36
Using a CellFormat Object for a Data Cell	27-37
Specifying a Cell Format	27-37
Configuring Stoplight and Conditional Formatting Using CellFormat	27-39

28 Using Gantt Chart Components

About the Gantt Chart Components	28-1
Gantt Chart Component Use Cases and Examples	28-2
End User and Presentation Features	28-3
Gantt Chart Regions	28-3
Information Panel	28-4
Toolbar	28-4
Scrolling, Zooming, and Panning	28-6
Showing Dependencies	28-7
Context Menus	28-8
Row Selection	28-9
Editing Tasks	28-10
Server-Side Events	28-10
Printing	28-11
Content Delivery	28-11
Additional Functionality for Gantt Chart Components	28-11
Using the Gantt Chart Components	28-12
Data for a Project Gantt Chart	28-12
Data for a Resource Utilization Gantt Chart	28-14
Data for a Scheduling Gantt Chart	28-15
Gantt Chart Tasks and Resources	28-17
Configuring Gantt Charts	28-18
How to Add a Gantt Chart to a Page	28-19
What Happens When You Add a Gantt Chart to a Page	28-22

Customizing Gantt Chart Tasks and Resources	28-23
Creating a New Task Type	28-23
Configuring Stacked Bars in Resource Utilization Gantt Charts	28-24
Configuring a Resource Capacity Line	28-25
Displaying Resource Attribute Details	28-25
Configuring Background Bars in Scheduling Gantt Charts	28-27
Customizing Gantt Chart Display Elements	28-30
Customizing Gantt Chart Toolbars and Menus	28-30
Creating Custom Toolbar and Menu Items	28-33
Customizing Gantt Chart Context Menus	28-34
How to Customize the Time Axis of a Gantt Chart	28-37
Creating and Customizing a Gantt Chart Legend	28-39
How to Specify Custom Data Filters	28-40
Specifying Nonworking Days in a Gantt Chart	28-41
How to Specify Weekdays as Nonworking Days	28-41
How to Identify Specific Dates as Nonworking Days	28-42
How to Apply Read-Only Values to Gantt Chart Features	28-42
What You May Need to Know About Skinning and Customizing the Appearance of Gantt Charts	28-46
Adding Interactive Features to Gantt Charts	28-46
Performing an Action on Selected Tasks or Resources	28-46
Using Page Controls for a Gantt Chart	28-47
Configuring Synchronized Scrolling Between Gantt Charts	28-48
Printing a Gantt Chart	28-51
Print Options	28-51
Action Listener to Handle the Print Event	28-52
Adding a Double-Click Event to a Task Bar	28-53
Using Gantt Charts as a Drop Target or Drag Source	28-53

29 Using Timeline Components

About Timeline Components	29-1
Timeline Use Cases and Examples	29-1
End User and Presentation Features	29-2
Timeline Overview Options	29-2
Layout Options	29-3
Timeline Item Selection	29-4
Content Delivery	29-4
Timeline Image Formats	29-5
Timeline Display in Printable or E-mailable Pages	29-5
Additional Functionality for Timeline Components	29-6
Using Timeline Components	29-7

Timeline Component Data Requirements	29-7
Configuring Timelines	29-9
How to Add a Timeline to a Page	29-10
What Happens When You Add a Timeline to a Page	29-12
Adding Data to Timeline Components	29-13
How to Add Data to a Timeline	29-13
What You May Need to Know About Configuring Data for a Dual Timeline	29-15
What You May Need to Know About Adding Data to Timelines	29-16
Customizing Timeline Display Elements	29-16
Customizing Timeline Items	29-17
Configuring a Timeline Item Duration	29-17
How to Add a Custom Time Scale to a Timeline	29-18
What You May Need to Know About Skinning and Customizing the Appearance of Timelines	29-19
Adding Interactive Features to Timelines	29-19
How to Add Popups to Timeline Items	29-19
Configuring Timeline Context Menus	29-20

30 Using Map Components

About Map Components	30-1
Map Component Use Cases and Examples	30-1
End User and Presentation Features of Maps	30-5
Geographic Map End User and Presentation Features	30-5
Thematic Map End User and Presentation Features	30-8
Additional Functionality for Map Components	30-11
Using the Geographic Map Component	30-12
Configuring Geographic Map Components	30-12
How to Add a Geographic Map to a Page	30-14
What Happens When You Add a Geographic Map to a Page	30-17
What You May Need to Know About Active Data Support for Map Point Themes	30-18
Customizing Geographic Map Display Attributes	30-18
How to Adjust the Map Size	30-18
How to Specify Strategy for Map Zoom Control	30-19
How to Customize and Use Map Selections	30-20
How to Customize the Map Legend	30-22
What You May Need to Know About Skinning and Customizing the Appearance of Geographic Maps	30-23
Customizing Geographic Map Themes	30-24
How to Customize Zoom Levels for a Theme	30-24
How to Customize the Labels of a Map Theme	30-24
How to Customize Color Map Themes	30-25

How to Customize Point Images in a Point Theme	30-26
What Happens When You Customize the Point Images in a Map	30-27
How to Customize the Bars in a Bar Graph Theme	30-28
What Happens When You Customize the Bars in a Map Bar Graph Theme	30-29
How to Customize the Slices in a Pie Graph Theme	30-29
What Happens When You Customize the Slices in a Map Pie Graph Theme	30-30
Adding a Toolbar to a Geographic Map	30-30
How to Add a Toolbar to a Map	30-31
What Happens When You Add a Toolbar to a Map	30-31
Using Thematic Map Components	30-32
Configuring Thematic Maps	30-32
Using the Layer Browser	30-37
How to Add a Thematic Map to a Page	30-38
What Happens When You Add a Thematic Map to a Page	30-42
What You May Need to Know About Thematic Map Image Formats	30-43
Defining Thematic Map Base Maps	30-43
Using Prebuilt Base Maps	30-43
Defining a Custom Base Map Using Map Provider APIs	30-44
Defining a Custom Base Map Using Image Files	30-47
Customizing Thematic Map Display Attributes	30-50
How to Customize Thematic Map Labels	30-50
How to Configure Tooltips to Display Data	30-52
How to Format Numeric Data Values in Area and Marker Labels	30-53
How to Configure Thematic Map Data Zooming	30-54
How to Configure Invisible Area Layers	30-56
What You May Need to Know About Skinning and Customizing the Appearance of a Thematic Map	30-58
Adding Interactive Features to Thematic Maps	30-58
How to Configure Selection and Action Events in Thematic Maps	30-58
How to Add Popups to Thematic Map Areas and Markers	30-60
How to Configure Animation Effects	30-64
How to Add Drag and Drop to Thematic Map Components	30-65

31 Using Hierarchy Viewer Components

About Hierarchy Viewer Components	31-1
Hierarchy Viewer Use Cases and Examples	31-1
End User and Presentation Features	31-2
Layouts	31-2
Navigation	31-5
Panning	31-5
Control Panel	31-5

Printing	31-7
Bi-directional Support	31-7
State Management	31-7
Additional Functionality for Hierarchy Viewer Components	31-7
Using Hierarchy Viewer Components	31-8
Configuring Hierarchy Viewer Components	31-9
How to Add a Hierarchy Viewer to a Page	31-10
What Happens When You Add a Hierarchy Viewer to a Page	31-13
What You May Need to Know About Hierarchy Viewer Rendering and Image Formats	31-14
Managing Nodes in a Hierarchy Viewer	31-15
How to Specify Node Content	31-16
How to Configure the Controls on a Node	31-18
Specifying a Node Definition for an Accessor	31-20
Associating a Node Definition with a Particular Set of Data Rows	31-21
How to Specify Ancestor Levels for an Anchor Node	31-21
Using Panel Cards	31-22
How to Create a Panel Card	31-22
What Happens at Runtime: How the Panel Card Component Is Rendered	31-24
Configuring Navigation in a Hierarchy Viewer	31-24
How to Configure Upward Navigation in a Hierarchy Viewer	31-25
How to Configure Same-Level Navigation in a Hierarchy Viewer	31-25
What Happens When You Configure Same-Level Navigation in a Hierarchy Viewer	31-26
Customizing the Appearance of a Hierarchy Viewer	31-27
How to Adjust the Display Size and Styles of a Hierarchy Viewer	31-27
Including Images in a Hierarchy Viewer	31-28
How to Configure the Display of the Control Panel	31-29
How to Configure the Display of Links and Labels	31-30
How to Disable the Hover Detail Window	31-31
What You May Need to Know About Skinning and Customizing the Appearance of a Hierarchy Viewer	31-32
Adding Interactivity to a Hierarchy Viewer Component	31-32
How to Configure Node Selection Action	31-32
Configuring a Hierarchy Viewer to Invoke a Popup Window	31-33
Configuring Hierarchy Viewer Drag and Drop	31-34
How to Configure Hierarchy Viewer Drag and Drop	31-37
What You May Need to Know About Configuring Hierarchy Viewer Drag and Drop	31-45
Adding Search to a Hierarchy Viewer	31-45
How to Configure Searching in a Hierarchy Viewer	31-45

32 Using Treemap and Sunburst Components

About the Treemap and Sunburst Components	32-1
Treemap and Sunburst Use Cases and Examples	32-1
End User and Presentation Features of Treemaps and Sunbursts	32-3
Treemap and Sunburst Layouts	32-3
Attribute Groups	32-5
Legend Support	32-6
Pattern Support	32-6
Node Selection Support	32-7
Tooltip Support	32-8
Popup Support	32-8
Context Menus	32-9
Drilling Support	32-10
Other Node Support	32-13
Drag and Drop Support	32-14
Sorting Support	32-15
Treemap and Sunburst Image Formats	32-16
Advanced Node Content	32-16
Printing and Email Support	32-18
Active Data Support (ADS)	32-18
Isolation Support (Treemap Only)	32-18
Treemap Group Node Header Customization (Treemap Only)	32-19
Additional Functionality for Treemap and Sunburst Components	32-20
Using the Treemap and Sunburst Components	32-21
Treemap and Sunburst Data Requirements	32-22
Using the Treemap Component	32-23
Configuring Treemaps	32-23
How to Add a Treemap to a Page	32-24
What Happens When You Add a Treemap to a Page	32-26
Using the Sunburst Component	32-27
Configuring Sunbursts	32-27
How to Add a Sunburst to a Page	32-29
What Happens When You Add a Sunburst to a Page	32-31
Adding Data to Treemap and Sunburst Components	32-31
How to Add Data to Treemap or Sunburst Components	32-31
What You May Need to Know about Adding Data to Treemaps and Sunbursts	32-34
Customizing Treemap and Sunburst Display Elements	32-34
Configuring Treemap and Sunburst Display Size and Style	32-34

What You May Need to Know About Skinning and Configuring Treemap and Sunburst Display Size and Style	32-34
Configuring Pattern Display	32-35
Configuring Treemap and Sunburst Attribute Groups	32-35
How to Configure Treemap and Sunburst Discrete Attribute Groups	32-36
How to Configure Treemap or Sunburst Continuous Attribute Groups	32-39
What You May Need to Know About Configuring Attribute Groups	32-41
How to Configure Treemap and Sunburst Legends	32-41
Configuring the Treemap and Sunburst Other Node	32-42
How to Configure the Treemap and Sunburst Other Node	32-42
What You May Need to Know About Configuring the Treemap and Sunburst Other Node	32-45
Configuring Treemap and Sunburst Sorting	32-46
Configuring Treemap and Sunburst Advanced Node Content	32-46
How to Add Advanced Node Content to a Treemap	32-46
How to Add Advanced Root Node Content to a Sunburst:	32-47
What You May Need to Know About Configuring Advanced Node Content on Treemaps	32-48
How to Configure Animation in Treemaps and Sunbursts	32-48
Configuring Labels in Treemaps and Sunbursts	32-49
How to Configure Treemap Leaf Node Labels	32-49
How to Configure Sunburst Node Labels	32-50
Configuring Sunburst Node Radius	32-51
How to Configure a Sunburst Node Radius	32-52
What You May Need to Know about Configuring the Sunburst Node Radius	32-53
Configuring Treemap Node Headers and Group Gap Display	32-53
How to Configure Treemap Node Headers	32-53
What You May Need to Know About Treemap Node Headers	32-54
How to Customize Treemap Group Gaps	32-54
Adding Interactive Features to Treemaps and Sunbursts	32-55
Configuring Treemap and Sunburst Tooltips	32-55
Configuring Treemap and Sunburst Popups	32-56
How to Add Popups to Treemap and Sunburst Components	32-56
What You May Need to Know About Adding Popups to Treemaps and Sunburst Components	32-59
Configuring Treemap and Sunburst Selection Support	32-60
How to Add Selection Support to Treemap and Sunburst Components	32-60
What You May Need to Know About Adding Selection Support to Treemaps and Sunbursts	32-62
Configuring Treemap and Sunburst Context Menus	32-63
How to Configure Treemap and Sunburst Context Menus	32-63
What You May Need to Know About Configuring Treemap and Sunburst Context Menus	32-69

Configuring Treemap and Sunburst Drilling Support	32-70
How to Configure Treemap and Sunburst Drilling Support	32-70
What You May Need to Know About Treemaps and Drilling Support	32-71
How to Add Drag and Drop to Treemaps and Sunbursts	32-71
Configuring Isolation Support (Treemap Only)	32-77
How to Disable Isolation Support	32-78
What You May Need to Know About Treemaps and Isolation Support	32-78

33 Using Diagram Components

About the Diagram Component	33-1
Diagram Use Cases and Examples	33-1
End User and Presentation Features of Diagrams	33-4
Additional Functionality for Diagram Components	33-9
Using the Diagram Component	33-10
Diagram Data Requirements	33-11
Configuring Diagrams	33-11
What You May Need to Know About Using the Default Diagram Layout	33-14
How to Add a Diagram to a Page	33-15
What Happens When You Add a Diagram to a Page	33-17
How to Create Diagram Nodes	33-17
Using the Diagram Layout Framework	33-18
Layout Requirements and Processing	33-19
Configuring Diagram Layouts	33-20
How to Register a Custom Layout	33-20
Designing Simple Client Layouts	33-21

34 Using Tag Cloud Components

About the Tag Cloud Component	34-1
Tag Cloud Use Cases and Examples	34-1
End User and Presentation Features of Tag Clouds	34-2
Additional Functionality for Tag Cloud Components	34-4
Using the Tag Cloud Component	34-5
Tag Cloud Data Requirements	34-6
How to Add a Tag Cloud to a Page	34-6
What Happens When You Add a Tag Cloud to a Page	34-7
Configuring Tag Clouds	34-7

Part VI Completing Your View

35 Customizing the Appearance Using Styles and Skins

About Customizing the Appearance Using Styles and Skins	35-1
Customizing the Appearance Use Cases and Examples	35-3
Additional Functionality for Customizing the Appearance	35-4
Changing the Style Properties of a Component	35-4
How to Set an Inline Style	35-5
How to Set a Style Class	35-6
Enabling End Users to Change an Application's ADF Skin	35-7
How to Enable End Users Change an Application's ADF Skin	35-8
What Happens at Runtime: How End Users Change an Application's ADF Skin	35-9
Using Scalar Vector Graphics Image Files	35-9
What You May Need to Know About Inline SVG Support in ADF Faces	35-10

36 Internationalizing and Localizing Pages

About Internationalizing and Localizing ADF Faces Pages	36-1
Internationalizing and Localizing Pages Use Cases and Examples	36-2
Additional Functionality for Internationalizing and Localizing Pages	36-3
Using Automatic Resource Bundle Integration in JDeveloper	36-3
How to Set Resource Bundle Options	36-4
What Happens When You Set Resource Bundle Options	36-5
How to Create an Entry in a JDeveloper-Generated Resource Bundle	36-6
What Happens When You Create an Entry in a JDeveloper-Generated Resource Bundle	36-7
Manually Defining Resource Bundles and Locales	36-7
How to Create a Resource Bundle as a Property File or an XLIFF File	36-8
How to Create a Resource Bundle as a Java Class	36-10
How to Edit a Resource Bundle File	36-11
How to Register a Locale for Your Application	36-13
How to Register a Resource Bundle in Your Application	36-14
How to Use Resource Bundles in Your Application	36-15
What You May Need to Know About ADF Skins and Control Hints	36-16
What You May Need to Know About Overriding a Resource Bundle in a Customizable Application	36-16
Configuring Pages for an End User to Specify Locale at Runtime	36-17
How to Configure a Page for an End User to Specify Locale	36-17
What Happens When You Configure a Page to Specify Locale	36-19
What Happens at Runtime: How an End User Specifies a Locale	36-20
Configuring Optional ADF Faces Localization Properties	36-20
How to Configure Optional Localization Properties	36-21

37 Developing Accessible ADF Faces Pages

About Accessibility Support In ADF Faces	37-1
Additional Information for Accessibility Support in ADF Pages	37-2
Accessibility Support Guidelines at Sign-In	37-2
Specifying Component-Level Accessibility Properties	37-3
ADF Faces Component Accessibility Guidelines	37-3
Using ADF Faces Table Components with a Screen Reader	37-7
ADF Data Visualization Components Accessibility Guidelines	37-8
How to Define Access Keys for an ADF Faces Component	37-9
How to Define Localized Labels and Access Keys	37-10
Creating Accessible Pages	37-11
How to Use Partial Page Rendering	37-13
How to Use Scripting	37-13
How to Use Styles	37-16
How to Use Page Structures and Navigation	37-17
How to Use Images and Tables	37-18
How to Use WAI-ARIA Landmark Regions	37-19
Creating Accessible Active Data Components	37-20
How to Customize the Screen Reader Response for Scalar Active Data Components	37-23
How to Customize the Screen Reader Response for Collection-Based Active Data Components	37-24
What You May Need to Know About Pass-Through Attributes	37-27
Running Accessibility Audit Rules	37-30
How to Create an Audit Profile	37-30
How to Run Audit Report	37-31

38 Allowing User Customization on JSF Pages

About User Customization	38-1
User Customization Use Cases and Examples	38-8
Implementing Session Change Persistence	38-8
How to Implement Session Change Persistence	38-9
What Happens When You Configure Your Application to Use Change Persistence	38-9
What Happens at Runtime: How Changes are Persisted	38-9
What You May Need to Know About Using Change Persistence on Templates and Regions	38-9

39 Adding Drag and Drop Functionality

About Drag and Drop Functionality	39-1
Additional Functionality for Drag and Drop	39-4
Adding Drag and Drop Functionality for Attributes	39-5
How to add Drag and Drop Functionality	39-5
Adding Drag and Drop Functionality for Objects	39-6
How to Add Drag and Drop Functionality for a Single Object	39-7
What Happens at Runtime: How to Use Keyboard Modifiers	39-10
What You May Need to Know About Using the ClientDropListener	39-11
Adding Drag and Drop Functionality for Collections	39-11
How to Add Drag and Drop Functionality for Collections	39-12
What You May Need to Know About the dragDropEndListener	39-14
Adding Drag and Drop Functionality for Components	39-15
How to Add Drag and Drop Functionality for Components	39-16
Adding Drag and Drop Functionality Into and Out of a panelDashboard Component	39-17
How to Add Drag and Drop Functionality Into a panelDashboard Component	39-18
How to Add Drag and Drop Functionality Out of a panelDashboard Component	39-19
Adding Drag and Drop Functionality to a Calendar	39-21
How to Add Drag and Drop Functionality to a Calendar	39-21
What You May Need to Know About Dragging and Dropping in a Calendar	39-22
Adding Drag and Drop Functionality for DVT Components	39-23
Adding Drop Functionality for DVT Pareto and Stock Graphs	39-23
How to Add Drop Functionality to Pareto and Stock Graphs	39-23
Adding Drag and Drop Functionality for DVT Gantt Charts	39-25
How to Add Drag and Drop Functionality for a DVT Gantt Component	39-26
Adding Drag and Drop Functionality for DVT Hierarchy Viewers, Sunbursts, and Treemaps	39-29
Drag and Drop Example for DVT Hierarchy Viewers	39-29
Drag and Drop Example for DVT Sunbursts	39-30
Drag and Drop Example for DVT Treemaps	39-31
How to Add Drag and Drop Functionality for a DVT Hierarchy Viewer, Sunburst, or Treemap Component	39-32
Adding Drag and Drop Functionality for Timeline Components	39-37

40 Using Different Output Modes

About Using Different Output Modes	40-1
Output Mode Use Cases	40-2
Displaying a Page for Print	40-4
How to Use the showPrintablePageBehavior Tag	40-4
Creating Emailable Pages	40-5

How to Create an Emailable Page	40-6
How to Test the Rendering of a Page in an Email Client	40-7
What Happens at Runtime: How ADF Faces Converts JSF Pages to Emailable Pages	40-8

41 Using the Active Data Service with an Asynchronous Backend

About the Active Data Service	41-1
Active Data Service Use Cases and Examples	41-2
Process Overview for Using Active Data Service	41-2
Implementing the ActiveModel Interface in a Managed Bean	41-3
What You May Need to Know About Maintaining Read Consistency	41-6
What You May Need to Know About Navigating Away From the ADS Enabled Page	41-7
Passing the Event Into the Active Data Service	41-7
Registering the Data Update Event Listener	41-8
Configuring the ADF Component to Display Active Data	41-9

Part VII Appendices

A ADF Faces Configuration

About Configuring ADF Faces	A-1
Configuration in web.xml	A-1
How to Configure for JSF and ADF Faces in web.xml	A-2
What You May Need to Know About Required Elements in web.xml	A-3
What You May Need to Know About ADF Faces Context Parameters in web.xml	A-4
State Saving	A-5
Debugging	A-6
File Uploading	A-6
Save Query Mode	A-7
Resource Debug Mode	A-7
User Customization	A-8
Enabling the Application for Real User Experience Insight	A-8
Assertions	A-8
Dialog Prefix	A-9
Compression for CSS Class Names	A-9
Control Caching When You Have Multiple ADF Skins in an Application	A-9
Test Automation	A-9
UIViewRoot Caching	A-9
Themes and Tonal Styles	A-10

Partial Page Rendering	A-10
Partial Page Navigation	A-10
Postback Payload Size Optimization	A-11
JavaScript Partitioning	A-11
Framebusting	A-12
Version Number Information	A-14
Suppressing Auto-Generated Component IDs	A-14
ADF Faces Caching Filter	A-14
Configuring Native Browser Context Menus for Command Links	A-15
Internet Explorer Compatibility View Mode	A-16
Session Timeout Warning	A-16
JSP Tag Execution in HTTP Streaming	A-17
Clean URLs	A-17
Page Loading Splash Screen	A-17
Graph and Gauge Image Format	A-18
Geometry Management for Layout and Table Components	A-18
Rendering Tables Initially as Read Only	A-19
Scrollbar Behavior in Tables	A-19
Production Project Stage	A-20
Toggle Example Hints	A-21
What You May Need to Know About Other Context Parameters in web.xml	A-21
Configuration in faces-config.xml	A-22
How to Configure for ADF Faces in faces-config.xml	A-22
Configuration in adf-config.xml	A-24
How to Configure ADF Faces in adf-config.xml	A-24
Defining Caching Rules for ADF Faces Caching Filter	A-24
Configuring Flash as Component Output Format	A-26
Using Content Delivery Networks	A-27
What You May Need to Know About Skin Style Sheets and CDN	A-30
What You May Need to Know About Preparing Your Resource Files for CDNs	A-30
Configuration in adf-settings.xml	A-31
How to Configure for ADF Faces in adf-settings.xml	A-32
What You May Need to Know About Elements in adf-settings.xml	A-32
Help System	A-33
Caching Rules	A-33
Configuration in trinidad-config.xml	A-34
How to Configure ADF Faces Features in trinidad-config.xml	A-35
What You May Need to Know About Elements in trinidad-config.xml	A-36
Animation Enabled	A-36
Skin Family	A-36

Time Zone and Year	A-37
Enhanced Debugging Output	A-37
Page Accessibility Level	A-37
Language Reading Direction	A-38
Currency Code and Separators for Number Groups and Decimal Points	A-38
Formatting Dates and Numbers Locale	A-39
Output Mode	A-39
Number of Active PageFlowScope Instances	A-39
File Uploading	A-39
Custom File Uploaded Processor	A-40
Client-Side Validation and Conversion	A-40
What You May Need to Know About Configuring a System Property	A-40
Configuration in trinidad-skins.xml	A-41
Using the RequestContext EL Implicit Object	A-41
Performance Tuning	A-44

B Message Keys for Converter and Validator Messages

About ADF Faces Default Messages	B-1
Message Keys and Setter Methods	B-2
Converter and Validator Message Keys and Setter Methods	B-2
af:convertColor	B-2
af:convertDateTime	B-2
af:convertNumber	B-3
af:validateByteLength	B-5
af:validateDateRestriction	B-5
af:validateDateTimeRange	B-6
af:validateDoubleRange	B-7
af:validateLength	B-8
af:validateRegExp	B-9

C Keyboard Shortcuts

About Keyboard Shortcuts	C-1
Tab Traversal	C-2
Tab Traversal Sequence on a Page	C-2
Tab Traversal Sequence in a Table	C-3
Shortcut Keys	C-5
Accelerator Keys	C-5
Access Keys	C-7
Shortcut Keys for Common Components	C-9

Shortcut Keys for Widgets	C-10
Shortcut Keys for Rich Text Editor Component	C-11
Shortcut Keys for Table, Tree, and Tree Table Components	C-12
Shortcut Keys for ADF Data Visualization Components	C-15
Shortcut Keys for Calendar Component	C-22
Default Cursor or Focus Placement	C-24
The Enter Key	C-25

D Creating Web Applications for Touch Devices Using ADF Faces

About Creating Web Applications for Touch Devices Using ADF Faces	D-1
How ADF Faces Behaves in Mobile Browsers on Touch Devices	D-1
Best Practices When Using ADF Faces Components in a Mobile Browser	D-6

E Quick Start Layout Themes

F Code Samples

Samples for Chapter 4, "Using ADF Faces Client-Side Architecture"	F-1
The <code>adf-js-partitions.xml</code> File	F-1
Samples for Chapter 29, "Using Map Components"	F-8
Sample Code for Thematic Map Custom Base Map	F-8
Sample Code for Thematic Map Custom Base Map Area Layer	F-13
Samples for Chapter 31, "Using Treemap and Sunburst Components"	F-14
Sample Code for Treemap and Sunburst Census Data Example	F-14
Code Sample for Sunburst Managed Bean	F-20
Samples for Chapter 32, "Using Diagram Components"	F-21
Code Sample for Default Client Layout	F-22
Code Sample for Simple Circle Layout	F-27
Code Sample for Simple Vertical Layout	F-30

G Troubleshooting ADF Faces

About Troubleshooting ADF Faces	G-1
Getting Started with Troubleshooting the View Layer of an ADF Application	G-2
Using Test Automation for ADF Faces	G-5
Enabling Test Automation for ADF Faces	G-5
Simulating Mouse Events in ADF Faces Test Automation	G-7
Resolving Common Problems	G-9
Application Displays an Unexpected White Background	G-9
Application is Missing Expected Images	G-9

ADF Skin Does Not Render Properly	G-10
ADF Data Visualization Components Fail to Display as Expected	G-10
High Availability Application Displays a NotSerializableException	G-10
Unable to Reproduce Problem in All Web Browsers	G-11
Application is Missing Content	G-12
Browser Displays an ADF_Faces-60098 Error	G-12
Browser Displays an HTTP 404 or 500 Error	G-12
Browser Fails to Navigate Between Pages	G-13
Using My Oracle Support for Additional Troubleshooting Information	G-13

Preface

Welcome to *Developing Web User Interfaces with Oracle ADF Faces!*

Audience

This document is intended for developers who need to create the view layer of a web application using the rich functionality of ADF Faces components.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Related Documents

For more information, see the following related documents:

- *Developing ADF Skins*
- *Understanding Oracle Application Development Framework*
- *Developing Applications with Oracle JDeveloper*
- *Developing Fusion Web Applications with Oracle Application Development Framework*
- *Administering Oracle ADF Applications*
- *Developing Applications with Oracle ADF Desktop Integration*
- *Java API Reference for Oracle ADF Faces*
- *Java API Reference for Oracle ADF Data Visualization Components*
- *JavaScript API Reference for Oracle ADF Faces*
- *Tag Reference for Oracle ADF Faces*
- *Tag Reference for Oracle ADF Faces Skin Selectors*
- *Tag Reference for Oracle ADF Faces Data Visualization Tools*
- *Java API Reference for Oracle ADF Lifecycle*

- *Java API Reference for Oracle ADF Resource Bundle*
- *Oracle JDeveloper 12c Release Notes*, included with your JDeveloper 12c installation, and on Oracle Technology Network

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide for Release 12c (12.2.1.4.0)

The following topics introduce the new and changed features of ADF Faces and other significant changes, which are described in this guide.

New and Changed Features for Release 12c (12.2.1.4.0)

Oracle Fusion Middleware Release 12c (12.2.1.4.0) of Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) includes the following new and changed development features, which are described in this guide.

- Common ADF Faces Components
 - ADF Faces supports filtering from a list of values to search and highlight the matched suggestion by using the new `inputSearch` component. You must use the ADF `inputSearch` component with REST resource and `inputSearch` supports tag attributes to understand REST data. See [Using the InputSearch Component](#).
 - ADF Faces supports a new streaming component in a JSF page. It allows the components to delay their rendering until a streaming request is fulfilled. See [Using a Streaming Component to Allow Page Loading](#).
 - ADF Faces supports most recently used (MRU) suggestions lists with the new `suggestionSection` component when used on a parent `inputSearch` component. The component creates and manages the MRU suggestion list based on the frequency and recentness of the suggestions used. See [How to Set Attributes of Suggestion Section](#).
 - ADF Faces supports new query component options to configure personalization behavior for the Saved Search dropdown to optimize performance for searches over all records. Also, you can implement an event handler for more query operation actions, including include saving a search, personalizing a search, and resetting a search. See [Creating the Query Data Model](#).
 - ADF Faces supports a new save query mode on the query component to prevent users from creating a new search, updating an existing search, or deleting an existing saved search, when the user is in preview mode. See [Save Query Mode](#).
 - ADF Faces supports a vertical compressed layout on the `panelTabbed` component. See [What You May Need to Know About Skinning and the panelTabbed Component](#).
 - ADF Faces supports using inline Scalar Vector Graphics (SVG) images via CSS and Javascript for ADF Faces components to access various elements available within the SVG DOM. See [Using Scalar Vector Graphics Image Files](#). Additionally, ADF supports styling inline Scalar Vector Graphics (SVG) by adding a style class in skin file and referencing it in the SVG content. See [What You May Need to Know About Inline SVG Support in ADF Faces](#).
 - ADF Faces supports configuring the `inputDate` component to show a default value for either the date or both the date and the time. Also, you can now

specify the date picker functionality to expose and to optionally specify the number of months to be shown at once in the date picker. See [What You May Need to Know About Including a Default Value for an InputDate Component](#) and [How to Add an InputDate Component](#).

- ADF Faces supports implementing event handlers to handle data selection in the chooseDate component based on selection modifier keys (Ctrl and Shift keys). See [What You May Need to Know About Multi-Selection Support in the chooseDate Component](#).
- ADF Faces supports new `web.xml` context parameters to configure the session timeout to override window timeout and to set a maximum window cache size. See [What You May Need to Know About ADF Faces Window Manager Configuration](#).
- Creating Your Layout
 - ADF supports configuring the `panelFormLayout` component to be responsive so that it modifies the form layout dynamically, depending on the space available. See [What You May Need to Know About Responsive Mode in the panelFormLayout Component](#).
 - ADF supports accessing a page from a different viewID by using a new public interface `QueryResultsLayoutIdentifier` to map a logical viewID to an actual viewID. See [What You May Need to Know About Accessing a Page From a Different View ID](#).
- ADF Data Converters
 - ADF Faces supports defining value formatting for a table or chart with Active Data Service (ADS) data by using the new `oracle.adf.view.rich.ads.USE_COMPONENT_FORMATTER` context parameter in the `web.xml` file. See [What You May Need to Know About Custom ADF Faces Converters](#).

Other Significant Changes in this Document for Release 12c (12.2.1.4.0)

For Release 12c (12.2.1.4.0), this document has been updated in several ways. Following are the sections that have been added or changed.

Part III Creating Your Layout

- Revised a `showDetailHeader` component section to explain that if `disclosed` property is set to `false` (for a collapsed state), then any validation present for child components will not be performed as the child components are not visible. See [How to Use the showDetailHeader Component](#).

Part IV Using Common ADF Faces Components

- Revised an LOV component section to carry forward the value entered in the input field of `inputListOfValues` and `inputComboboxListOfValues` components to the appropriate search field in the search dialog. See [About List-of-Values Components](#).
- Revised a `listView` component section to note that you must include the `fetchSize` and `rows` attributes on the component for the `matchMediaBehavior` tag to work properly. See [Achieving Responsive Behavior Using matchMediaBehavior Tag](#).

Part VI Completing Your View

- Revised an accessibility section for active data service (ADS) components to remove the `type=email` attribute on the input fields in the generated DOM sample. Also added a note to explain that the passthrough attribute is stamped on the outermost element. See [What You May Need to Know About Pass-Through Attributes](#).

Part VII Appendices

- Revised a keyboard shortcuts section to include Ctrl+right arrow and Ctrl+ left arrow to expand and collapse nodes in the TreeTable component. Also removed references to Screen Reader Mode. See [Shortcut Keys for Table, Tree, and Tree Table Components](#).

Part I

Getting Started with ADF Faces

Part I introduces you to the ADF Faces components and their architecture, the ADF Faces Components Demo application, and how to start developing with ADF Faces using JDeveloper.

Specifically, this part contains the following chapters:

- [Introduction to ADF Faces](#)
- [ADF Faces Components Demo Application](#)
- [Getting Started with ADF Faces and JDeveloper](#)

1

Introduction to ADF Faces

This chapter introduces ADF Faces, providing an overview of the framework functionality and each of the different component types found in the library.

This chapter includes the following sections:

- [About ADF Faces](#)
- [ADF Faces Framework](#)
- [ADF Faces Components](#)

For definitions of unfamiliar terms found in this and other books, see the Glossary.

About ADF Faces

ADF Faces is the view/controller part of the Oracle ADF end-to-end framework. ADF Faces has more than 150 Ajax-enabled components that help you to quickly build applications that are robust, responsive, and easy to use.

ADF Faces is a set of over 150 Ajax-enabled JavaServer Faces (JSF) components as well as a complete framework, all built on top of the JSF 2.0 standard. In its beginnings, ADF Faces was a first-generation set of JSF components, and has since been donated to the Apache Software Foundation. That set is now known as Apache MyFaces Trinidad (currently available through the Apache Software Foundation), and remains as the foundation of today's ADF Faces.

With ADF Faces and the advent of JSF 2.0, you can implement Ajax-based applications relatively easily with a minimal amount of hand-coded JavaScript. For example, you can easily build a stock trader's dashboard application that allows a stock analyst to use drag and drop to add new stock symbols to a table view, which then gets updated by the server model using an advanced push technology. To close new deals, the stock trader could navigate through the process of purchasing new stocks for a client, without having to leave the actual page. Much of this functionality can be implemented declaratively using Oracle JDeveloper, a full-featured development environment with built-in support for ADF Faces components, allowing you to quickly and easily build the view layer of your web application.

Note:

Because ADF Faces adheres to the standards of the JSF technology, this guide is mostly concerned with content that is in addition to, or different from, JSF standards. Therefore, you should have a basic understanding of how JSF works before beginning to develop with ADF Faces. To learn about JSF, see <http://www.oracle.com/technetwork/java/javasee/javaserverfaces-139869.html>.

ADF Faces Framework

ADF Faces is built on top of the JavaServer Faces (JSF) 2.0 framework that provides a commercial Java framework for building enterprise applications. It provides Java EE and web standards based application development in a productive (declarative, visual) way.

ADF Faces framework offers complete rich functionality, including the following:

- Built to the JavaServer Faces (JSF) 2.0 and later specification

Currently, ADF Faces supports JSF 2.2, including Facelets. Several of the JSF 2.0 features have parallel functionality in ADF Faces. To understand the functionality introduced in JSF 2.0 and the functional overlap that exists between ADF Faces and JSF 2.0, see the *JavaServer Faces 2.0 Overview and Adoption Roadmap in Oracle ADF Faces and Oracle JDeveloper 11g* whitepaper on OTN at <http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adffaces-jsf20-190927.pdf>.
- Large set of fully featured rich components that are optimized to run in browsers on a desktop or a tablet device.

The library provides over 150 Rich Internet Application (RIA) components, including geometry-managed layout components, text and selection components, sortable and hierarchical data tables and trees, menus, in-page dialogs, and general controls. See [ADF Faces Components](#). For information about running ADF Faces on tablets, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Widespread Ajax support

Many ADF Faces components have ajax-style functionality implemented natively. For example, the ADF Faces table component lets you scroll through the table, sort the table by clicking a column header, mark a row or several rows for selection, and even expand specific rows in the table, all without requiring the page to be submitted to the server, and with no coding needed. In ADF Faces, this functionality is implemented as **partial page rendering** (PPR), as described in [Rerendering Partial Page Content](#).
- Limited need for developers to write JavaScript

ADF Faces hides much of the complex JavaScript from you. Instead, you declaratively control how components function. You can implement a rich, functional, attractive web UI using ADF Faces in a declarative way that does not require the use of any JavaScript at all.

That said, there may be cases when you do want to add your own functionality to ADF Faces, and you can easily do that using the client-side component and event framework. See [Using ADF Faces Client-Side Architecture](#).
- Enhanced lifecycle on both server and client

ADF Faces extends the standard JSF 2.0 page request lifecycle. Examples include a client-side value lifecycle, a subform component that allows you to create independent submittable regions on a page without needing multiple forms, and an optimized lifecycle that can limit the parts of the page submitted for processing. See [Using the JSF Lifecycle with ADF Faces](#).
- Event handling

ADF Faces adheres to standard JSF event handling techniques, as well as offering complete a client-side event model. For information about events, see [Handling Events](#).

- Partial page navigation

ADF Faces applications can use PPR for navigation, which eliminates the need to repeatedly load JavaScript libraries and stylesheets when navigating between pages. See [Using Partial Page Navigation](#).

- Client-side validation, conversion, and messaging

ADF Faces validators can operate on both the client and server side. Client-side validators are written in JavaScript and validation errors caught on the client-side can be processed without a round-trip to the server. See [Validating and Converting Input](#).

- Server-side push and streaming

The ADF Faces framework includes server-side push that allows you to provide real-time data updates for ADF Faces components, as described in [Using the Active Data Service with an Asynchronous Backend](#).

- Active geometry management

ADF Faces provides a client-side geometry management facility that allows components to determine how best to make use of available screen real-estate. The framework notifies layout components of browser resize activity, and they in turn are able to resize their children. This allows certain components to stretch or shrink, filling up any available browser space. See [Geometry Management and Component Stretching](#).

- Advanced templating and declarative components

You can create page templates, as well as page fragments and composite components made up of multiple components, which can be used throughout your application, as described in [Creating and Reusing Fragments, Page Templates, and Components](#).

- Advanced visualization components

ADF Faces includes data visualization components, which are Flash- and PNG-enabled components capable of rendering dynamic charts, graphs, gauges, and other graphics that provide a real-time view of underlying data. See [Using ADF Data Visualization Components](#).

- Skinning

You can create your own look and feel by implementing skins for ADF Faces components. Oracle provides tools, where you can declaratively create and modify your skins. See [Customizing the Appearance Using Styles and Skins](#).

- Output modes

You can make it so that pages that normally display in an HTML browser can be displayed in another mode, such as email or print view, as described in [Using Different Output Modes](#).

- Internationalization

You can configure your JSF page or application to use different locales so that it displays the correct language based on the language setting of a user's browser, as described in [Internationalizing and Localizing Pages](#).

- Accessibility

ADF Faces components have built-in accessibility that work with a range of assistive technologies, including screen readers. ADF Faces accessibility audit rules provide direction to create accessible images, tables, frames, forms, error messages, and popup windows using accessible HTML markup. See [Developing Accessible ADF Faces Pages](#).

- User-driven personalization

Many ADF Faces components allow users to change the display of the component at runtime. By default, these changes live only as long as the page request. However, you can configure your application so that the changes can be persisted through the length of the user's session. See [Allowing User Customization on JSF Pages](#).

- Drag and drop

The ADF Faces framework allows the user to move data from one location to another by dragging and dropping one component onto another, as described in [Adding Drag and Drop Functionality](#).

- Integration with other Oracle ADF technologies

You can use ADF Faces in conjunction with the other Oracle ADF technologies, including ADF Business Components, ADF Controller, and ADF data binding. For information about using ADF Faces with the ADF technology stack, see *Creating a Databound Web User Interface in [Developing Fusion Web Applications with Oracle Application Development Framework](#)*.

- Integrated declarative development with Oracle JDeveloper

JDeveloper is a full-featured development environment with built-in declarative support for ADF Faces components, including a visual layout editor, a Components window that allows you to drag and drop an ADF Faces component onto a page, and a Properties window where you declaratively configure component functionality. For information about using JDeveloper, see [Getting Started with ADF Faces and JDeveloper](#).

ADF Faces Components

The ADF Faces components provide numerous user-interface additives with built-in functionality such as layout, backgrounds, data views, color, and so on that can be customized and re-used in your application.

ADF Faces components generally fall into the following categories:

- Layout components

Layout components act as containers to determine the layout of the page, ADF Faces layout components also include interactive container components that can show or hide content, or that provide sections, lists, or empty spaces. JDeveloper provides prebuilt quick-start layouts that declaratively add layout components to your page based on how you want the page to look. For information about layout components and geometry management, see [Organizing Content on Web Pages](#).

In addition to standard layout components, ADF Faces also provides the following specialty layout components:

- Explorer-type menus and toolbar containers: Allow you to create menu bars and toolbars. Menus and toolbars allow users to select from a specified list of

- options (in the case of a menu) or buttons (in the case of a toolbar) to cause some change to the application. See [Using Menus, Toolbars, and Toolboxes](#).
- Secondary windows: Display data in popup windows or dialogs. The dialog framework in ADF Faces provides an infrastructure to support building pages for a process displayed in a new popup browser window separate from the parent page. Multiple dialogs can have a control flow of their own. See [Using Popup Dialogs, Menus, and Windows](#).
- Core structure components and tags: Provide the tags needed to create pages and layouts, such as documents, forms and subforms, and resources. These tags are discussed in various chapters.
- Text and selection components

These components allow you to display text, from a simple output text component to input components, including selection components, to a complex list of value component.

 - Output components: Display text and graphics, and can also play video and music clips. ADF Faces also includes a carousel component that can display graphics in a revolving carousel. See [Using Output Components](#).
 - Input components: Allow users to enter data or other types of information, such as color selection or date selection. ADF Faces also provides simple lists from which users can choose the data to be posted, as well as a file upload component. For information about input components, see [Using Input Components and Defining Forms](#).
 - List-of-Values (LOV) components: Allow users to make selections from lists driven by a model that contains functionality like searching for a specific value or showing values marked as favorites. These LOV components are useful when a field used to populate an attribute for one object might actually be contained in a list of other objects, as with a foreign key relationship in a database. See [Using List-of-Values Components](#).
- Data Views

ADF Faces provides a number of different ways to display complex data.

 - Table and tree components: Display structured data in tables or expandable trees. ADF Faces tables provide functionality such as sorting column data, filtering data, and showing and hiding detailed content for a row. Trees have built-in expand/collapse behavior. Tree tables combine the functionality of tables with the data hierarchy functionality of trees. See [Using Tables, Trees, and Other Collection-Based Components](#).
 - Data visualization components: Allow users to view and analyze complex data in real time. ADF Data Visualization components include graphs, gauges, pivot tables, timelines, geographic and thematic maps, Gantt charts, hierarchy viewers, and treemap and sunbursts that display row set and hierarchical data, for example an organization chart. See [Introduction to ADF Data Visualization Components](#).
 - Query components: Allow users to query data. The query component can support multiple search criteria, dynamically adding and deleting criteria, selectable search operators, match all/any selections, seeded or saved searches, a basic or advanced mode, and personalization of searches. See [Using Query Components](#).
 - Specialty display components: The calendar component displays activities in day, week, month, or list view. You can implement popup components that

allow users to create, edit, or delete activities. See [Using a Calendar Component](#). The carousel component allows you to display a collection of images in a scrollable manner. See [Displaying Images in a Carousel](#).

- Messaging and help: The framework provides the ability to display tooltips, messages, and help for input components, as well as the ability to display global messages for the application. The help framework allows you to create messages that can be reused throughout the application. You create a help provider using a Java class, a managed bean, an XLIFF file, or a standard properties file, or you can link to an external HTML-based help system. See [Displaying Tips, Messages, and Help](#).
- Hierarchical menu model: ADF Faces provides navigation components that render items such as tabs and breadcrumbs for navigating hierarchical pages. The framework provides an XML-based menu model that, in conjunction with a metadata file, contains all the information for generating the appropriate number of hierarchical levels on each page, and the navigation items that belong to each level. See [Working with Navigation Components](#).
- General controls
General controls include the components used to navigate, as well as to display images and icons,
 - Navigation components: Allow users to go from one page to the next. ADF Faces navigation components include buttons and links, as well as the capability to create more complex hierarchical page flows accessed through different levels of menus. See [Working with Navigation Components](#).
 - Images and icon components: Allow you to display images as simple as icons, to as complex as video. See [Using Output Components](#).
- Operations
While not components, these tags work with components to provide additional functionality, such as drag and drop, validation, and a variety of event listeners. These operational tags are discussed with the components that use them.

2

ADF Faces Components Demo Application

This chapter describes the ADF Faces Components Demo application that can be used in conjunction with this developers guide.

This chapter contains the following sections:

- [About the ADF Faces Components Demo Application](#)
- [Downloading and Installing the ADF Faces Components Demo Application](#)

About the ADF Faces Components Demo Application

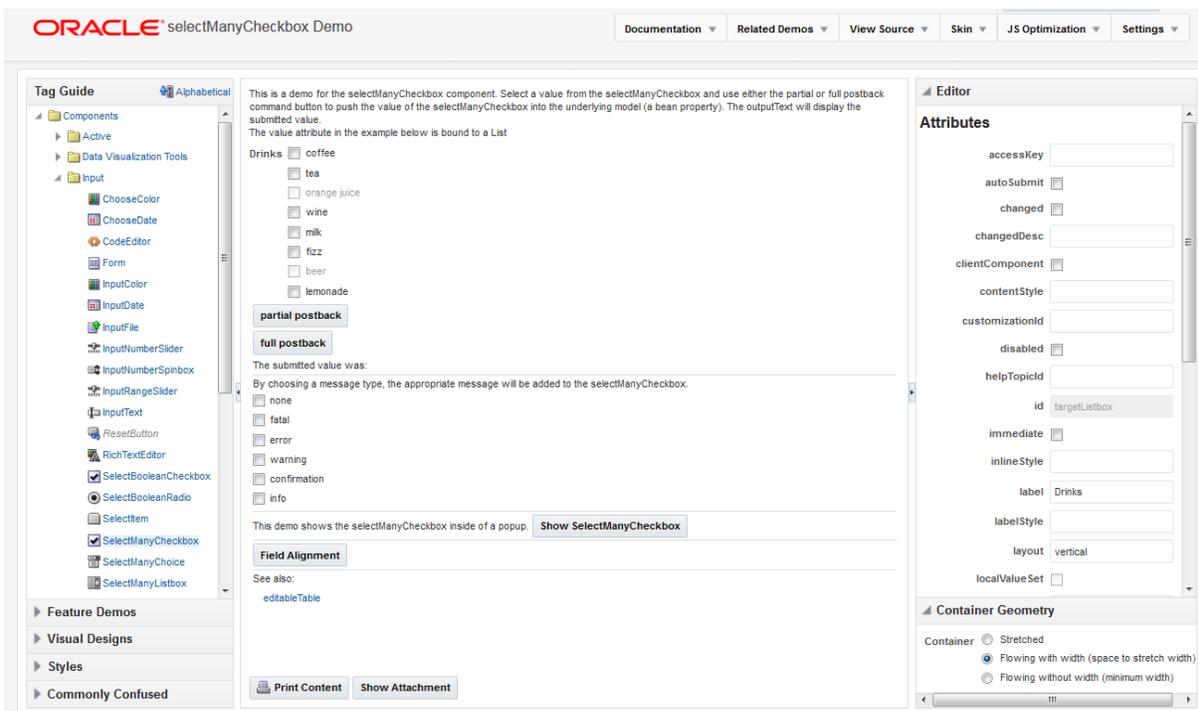
The ADF Faces Components demo application showcases the various components and framework capabilities and allows users to try different property settings on the selected component. The components demo is provided with full source code.

ADF Faces includes the ADF Faces Components Demo application that allows you both to experiment with running samples of the components and architecture features and to view the source code.

The ADF Faces Components Demo application contains the following:

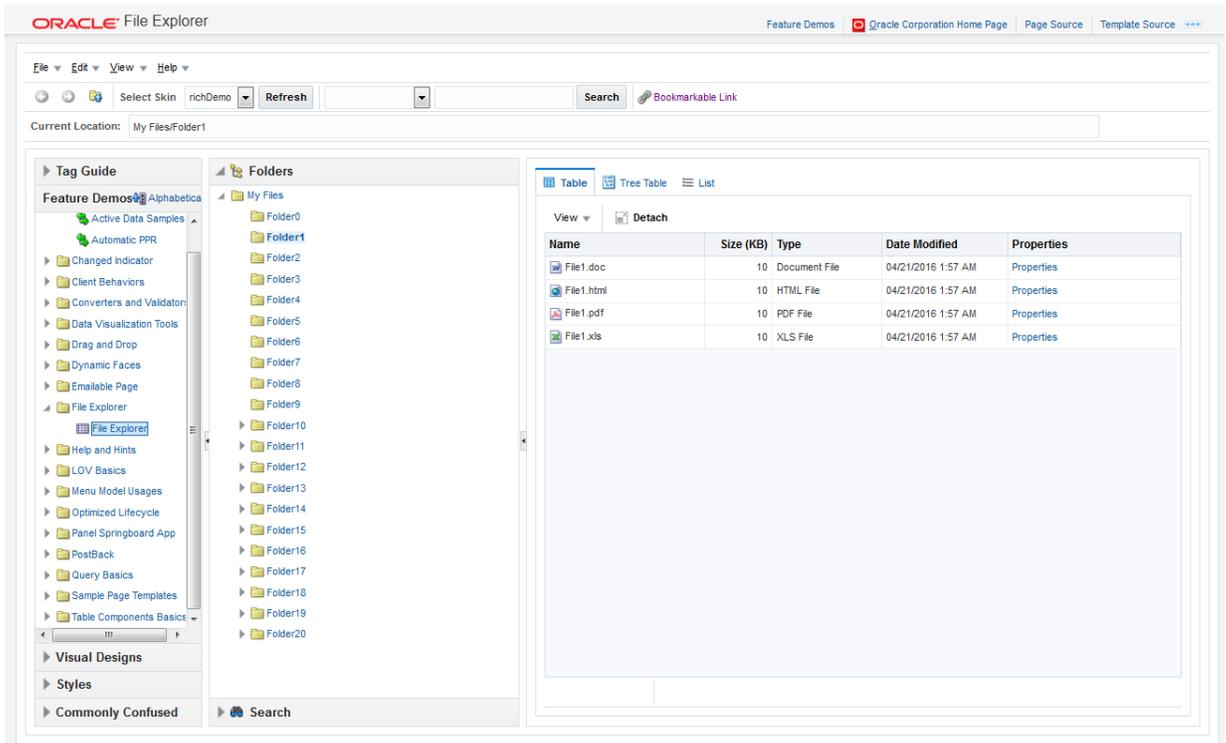
- Tag guide: Demonstrations of ADF Faces components, validators, converters, and miscellaneous tags, along with a property editor to see how changing attribute values affects the component. [Figure 2-1](#) shows the demonstration of the `selectManyCheckbox` component. Each demo provides a link to the associated tag documentation.

Figure 2-1 Tag Demonstration



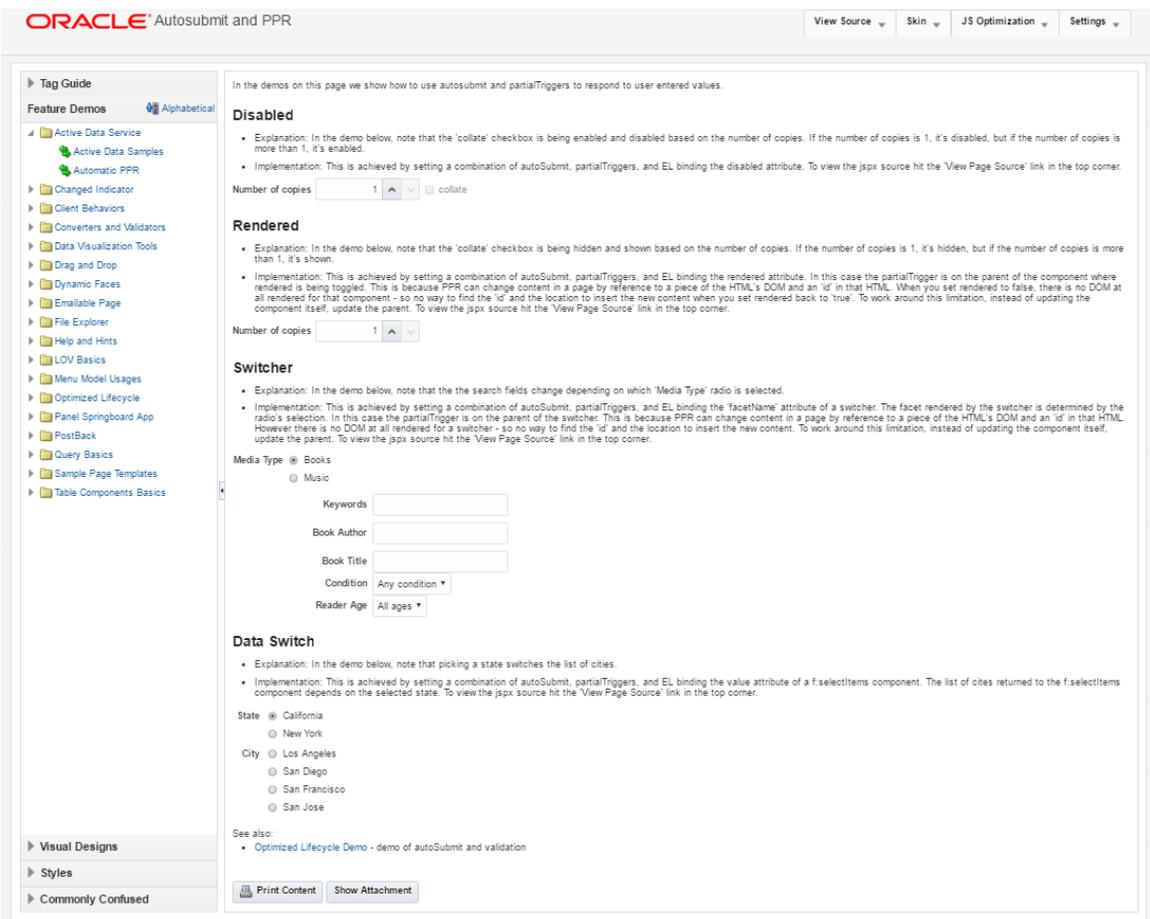
- Feature demos: Various pages that demonstrate different ways you can use ADF components. For example, the File Explorer is an application with a live data model that displays a directory structure and allows you to create, save, and move directories and files. This application is meant to showcase the components and features of ADF Faces in a working application, as shown in [Figure 2-2](#).

Figure 2-2 File Explorer Application



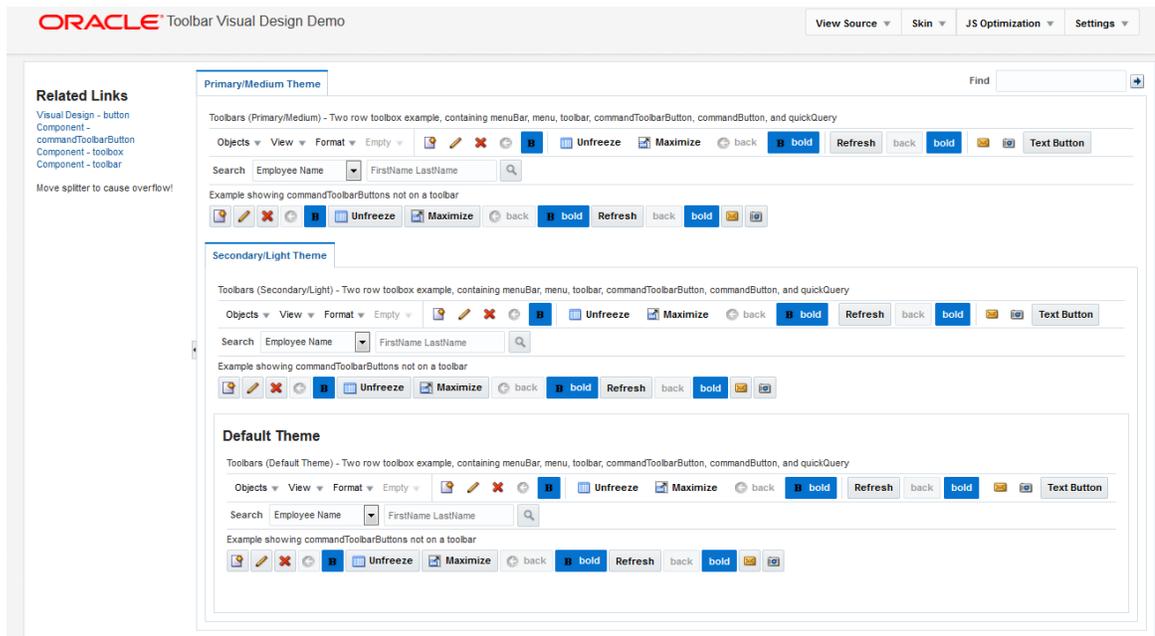
Other pages demonstrate the main architectural features of ADF Faces, such as layout components, Ajax postback functionality, and drag and drop. Figure 2-3 shows the demonstration on using the `AutoSubmit` attribute and partial page rendering.

Figure 2-3 Framework Demonstration



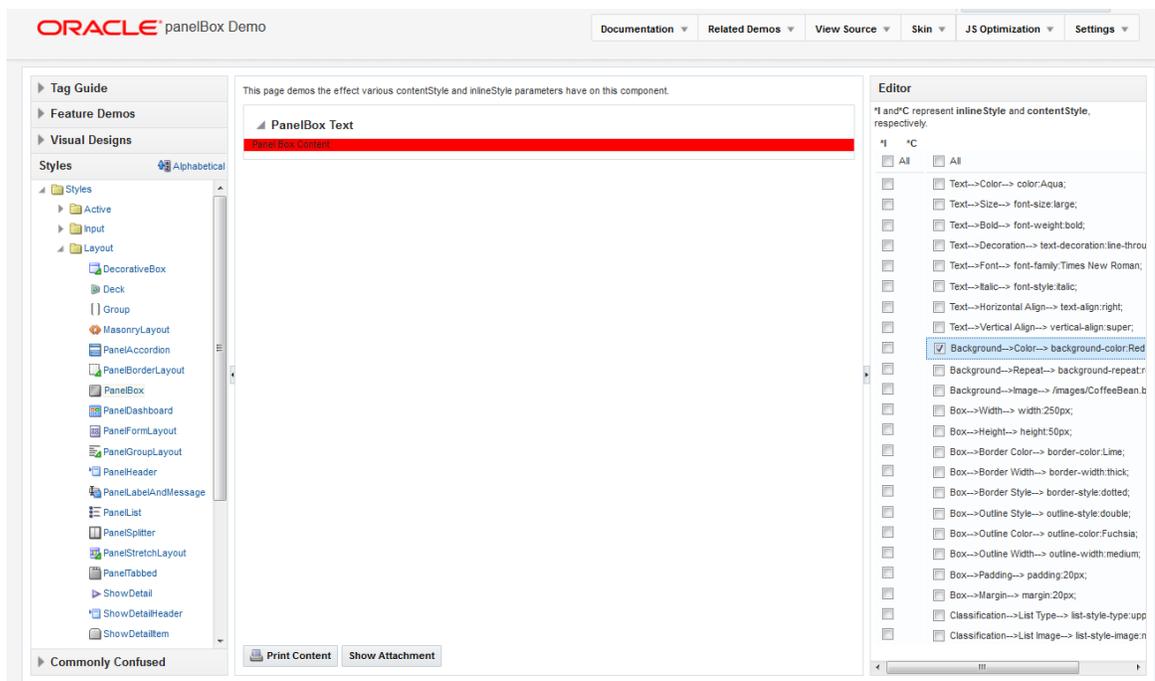
- Visual designs: Demonstrations of how you can use types of components in different ways to achieve different UI designs. Figure 2-4 shows how you can achieve different looks for a toolbar.

Figure 2-4 Toolbar Design Demonstration



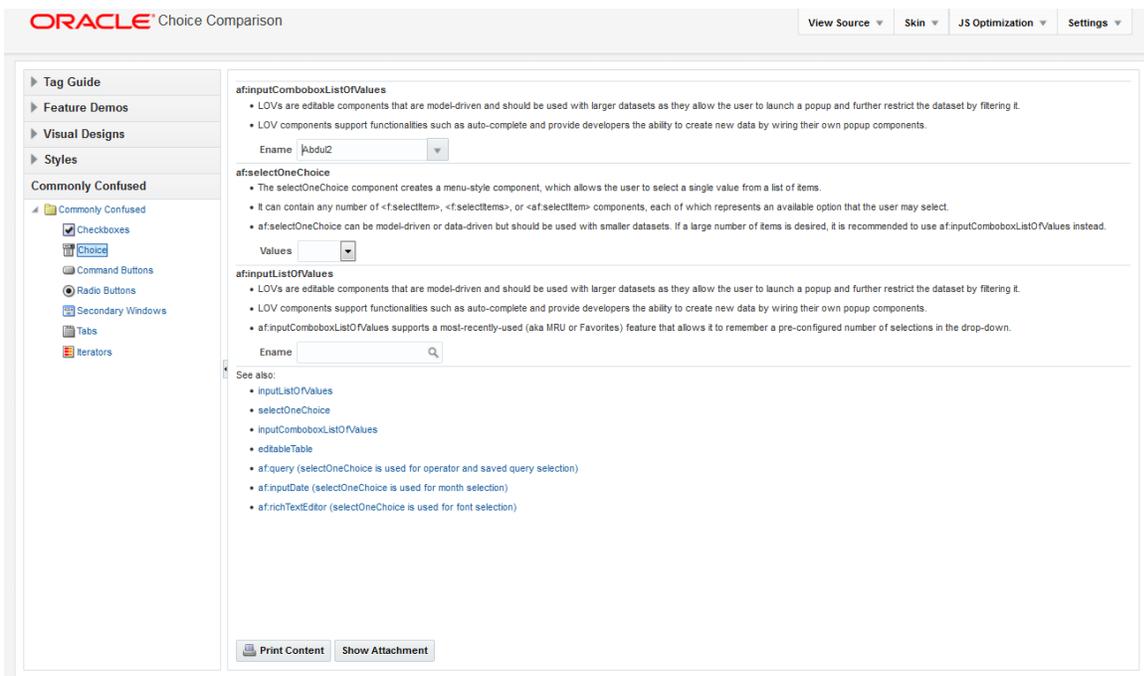
- Styles: Demonstration of how setting inline styles and content styles affects components. Figure 2-5 shows different styles applied to the panelBox component.

Figure 2-5 Styles Demonstration



- Commonly confused components: A comparison of components that provide similar functionality. Figure 2-6 shows the differences between the various components that display selection lists.

Figure 2-6 Commonly Confused Components



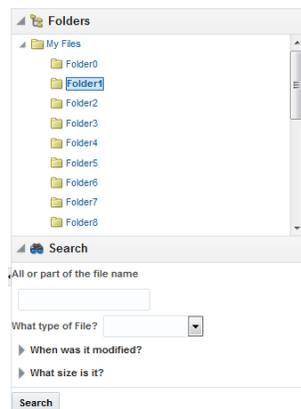
Additional Components of the File Explorer in the Demo Application

Because the File Explorer is a complete working application, many sections in this guide use that application to illustrate key points, or to provide code samples. The source for the File Explorer application can be found in the `fileExplorer` directory.

The File Explorer application uses the `fileExplorerTemplate` page template. This template contains a number of layout components that provide the basic look and feel for the application. For information about layout components, see [Organizing Content on Web Pages](#). For information about using templates, see [Creating and Reusing Fragments, Page Templates, and Components](#).

The left-hand side of the application contains a `panelAccordion` component that holds two areas: the directory structure and a search field with a results table, as shown in [Figure 2-7](#).

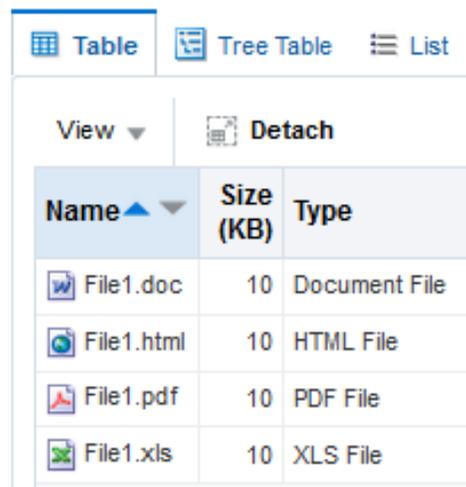
Figure 2-7 Directory Structure Panel and Search Panel



You can expand and collapse both these areas. The directory structure is created using a `tree` component. The search area is created using input components, a button, and a `table` component. For information about using `panelAccordion` components, see [Displaying or Hiding Contents in Panels](#). For information about using input components, see [Using Input Components and Defining Forms](#). For information about using buttons, see [Working with Navigation Components](#). For information about using tables and trees, see [Using Tables, Trees, and Other Collection-Based Components](#).

The right-hand side of the File Explorer application uses tabbed panes to display the contents of a directory in either a table, a tree table or a list, as shown in [Figure 2-8](#).

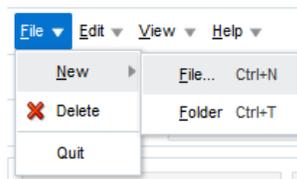
Figure 2-8 Directory Contents in Tabbed Panels



The table and tree table have built-in toolbars that allow you to manipulate how the contents are displayed. In the table and list, you can drag a file or subdirectory from one directory and drop it into another. In all tabs, you can right-click a file, and from the context menu, you can view the properties of the file in a popup window. For information about using tabbed panes, see [Displaying or Hiding Contents in Panels](#). For information about table and tree table toolbars, see [Displaying Table Menus, Toolbars, and Status Bars](#). For information about using context menus and popup windows, see [Using Popup Dialogs, Menus, and Windows](#).

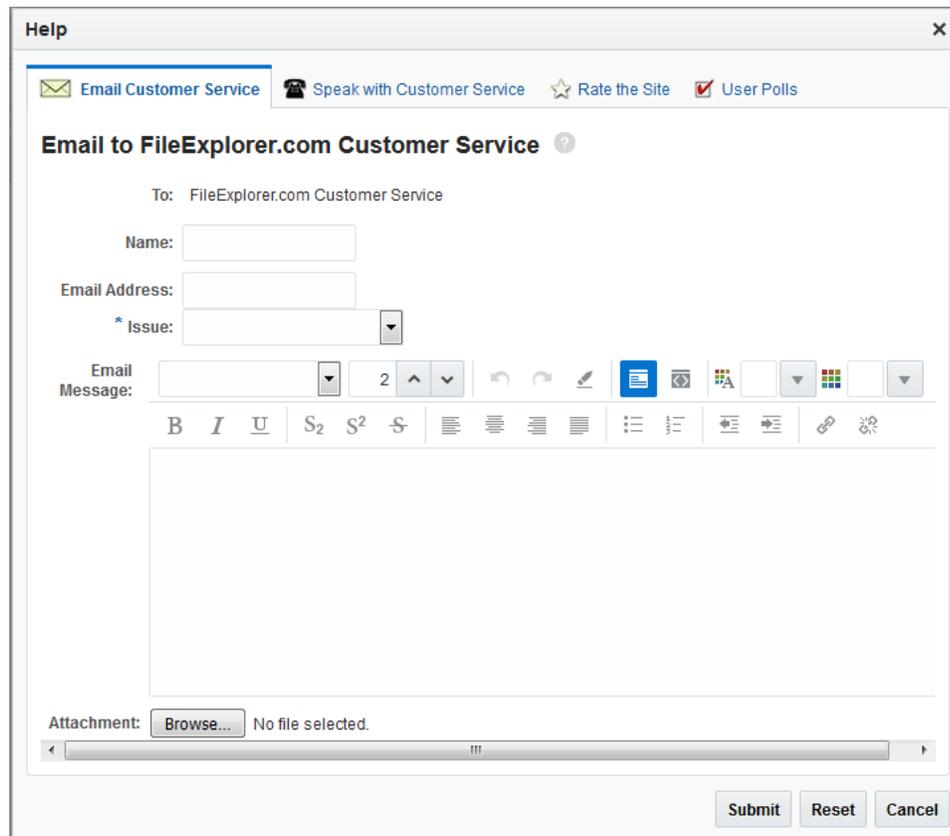
The top of the File Explorer application contains a menu and a toolbar, as shown in [Figure 2-9](#).

Figure 2-9 Menu and Toolbar



The menu options allow you to create and delete files and directories and change how the contents are displayed. The Help menu opens a help system that allows users to provide feedback in dialogs, as shown in [Figure 2-10](#).

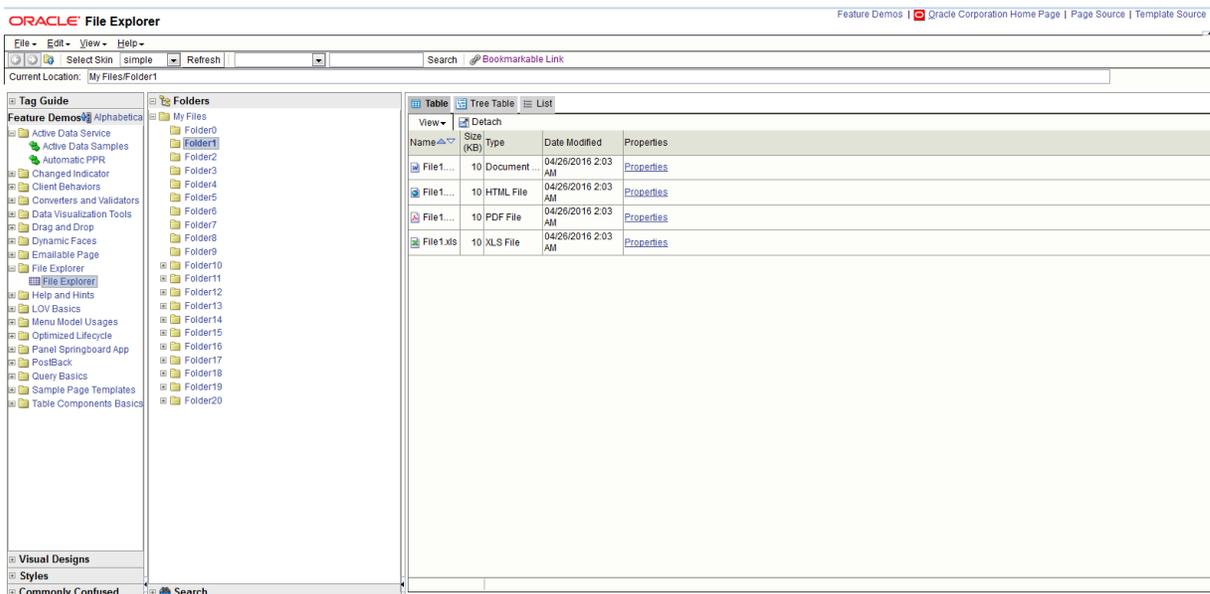
Figure 2-10 Help System



The help system consists of a number of forms created with various input components, including a rich text editor. For information about menus, see [Using Menus in a Menu Bar](#). For information about creating help systems, see [Displaying Help for Components](#). For information about input components, see [Using Input Components and Defining Forms](#).

Within the toolbar of the File Explorer are controls that allow you navigate within the directory structure, as well as controls that allow you to change the look and feel of the application by changing its skin. [Figure 2-11](#) shows the File Explorer application using the simple skin.

Figure 2-11 File Explorer Application with the Simple Skin



For information about toolbars, see [Using Toolbars](#). For information about using skins, see [Customizing the Appearance Using Styles and Skins](#).

Downloading and Installing the ADF Faces Components Demo Application

You can download and install the ADF Faces Components Demo application from Oracle Technology Network (OTN) web site; you can open and view the demo using JDeveloper when you want to view samples of ADF Faces components and to understand features.

In order to view the demo application (both the code and at runtime), install JDeveloper, and then download and open the application within JDeveloper.

You can download the ADF Faces Components Demo application from the Oracle Technology Network (OTN) web site. Navigate to <http://www.oracle.com/technetwork/developer-tools/adf/overview/index-092391.html> and click the **ADF Faces Components Demo** link in the **Download** section of the page. The resulting page provides detailed instructions for downloading the WAR file that contains the application, along with instructions for deploying the application to a standalone server, or for running the application using the Integrated WebLogic Server included with JDeveloper.

If you do not want to install the application, you can run the application directly from OTN by clicking the **ADF Faces Rich Client Components Hosted Demo** link.

3

Getting Started with ADF Faces and JDeveloper

This chapter describes how to use JDeveloper to declaratively create ADF Faces applications.

This chapter includes the following sections:

- [About Developing Declaratively in JDeveloper](#)
- [Creating an Application Workspace](#)
- [Defining Page Flows](#)
- [Creating a View Page](#)
- [Creating EL Expressions](#)
- [Creating and Using Managed Beans](#)
- [Viewing ADF Faces Javadoc](#)

About Developing Declaratively in JDeveloper

The ADF framework provides a visual and declarative approach to Java EE development. It supports rapid application development based on ready-to-use design patterns, metadata-driven and visual tools.

Using JDeveloper with ADF Faces and JSF provides a number of areas where page and managed bean code is generated for you declaratively, including creating EL expressions and automatic component binding. Additionally, there are a number of areas where XML metadata is generated for you declaratively, including metadata that controls navigation and configuration.

At a high level, the development process for an ADF Faces view project usually involves the following:

- [Creating an Application Workspace](#)
- [Defining Page Flows](#)
- [Creating a View Page](#) using either Facelets or JavaServer Pages (JSPs).
- Deploying the application. See *Deploying ADF Applications in Administering Oracle ADF Applications*. If your application uses ADF Faces with the ADF Model layer, ADF Controller, and ADF Business Components, see *Deploying Fusion Web Applications in Developing Fusion Web Applications with Oracle Application Development Framework*.

Ongoing tasks throughout the development cycle will likely include the following:

- [Creating and Using Managed Beans](#)
- [Creating EL Expressions](#)
- [Viewing ADF Faces Javadoc](#)

JDeveloper also includes debugging and testing capabilities. See Testing and Debugging ADF Components in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Creating an Application Workspace

An application workspace is a directory that helps the ADF application developer to gather various source code files and resources and work with them as a cohesive unit. It is the one of the first steps in building a new application.

The first steps in building a new application are to assign it a name and to specify the directory where its source files will be saved. You can either create an application that just contains the view layer, or you can add an ADF Faces project to an existing application.

Note:

This document covers only how to create the ADF Faces project in an application, without regard to the business services used or the binding to those services. For information about how to use ADF Faces with the ADF Model layer, ADF Controller, and ADF Business Components, see Getting Started with Your Web Interface in *Developing Fusion Web Applications with Oracle Application Development Framework*.

How to Create an ADF Faces Application Workspace

You create an application workspace using the Create Application wizard.

To create an application:

1. In the menu, choose **File > New > Application**.
2. In the New Gallery, select **Custom Application** and click **OK**.
3. In the Create Custom Application dialog, set a name, directory location, and package prefix of your choice and click **Next**.
4. In the Name Your Project page, you can optionally change the name and location for your view project. On the Project Features tab, shuttle **ADF Faces** to **Selected**. The necessary libraries and metadata files for ADF Faces will be added to your project. Click **Next**.
5. In the Configure Java Settings page, optionally change the package name, Java source path, and output directory for any Java classes you might create. Click **Finish**.

 **Tip:**

You can also add ADF Faces to an existing project (for example, a view project in a JEE web application). To do so:

- a. Right-click the project and choose **Project Properties**.
- b. In the Project Properties dialog, select **Features**, then click the **Add** (green plus) icon, and shuttle ADF Faces to the **Selected** pane.

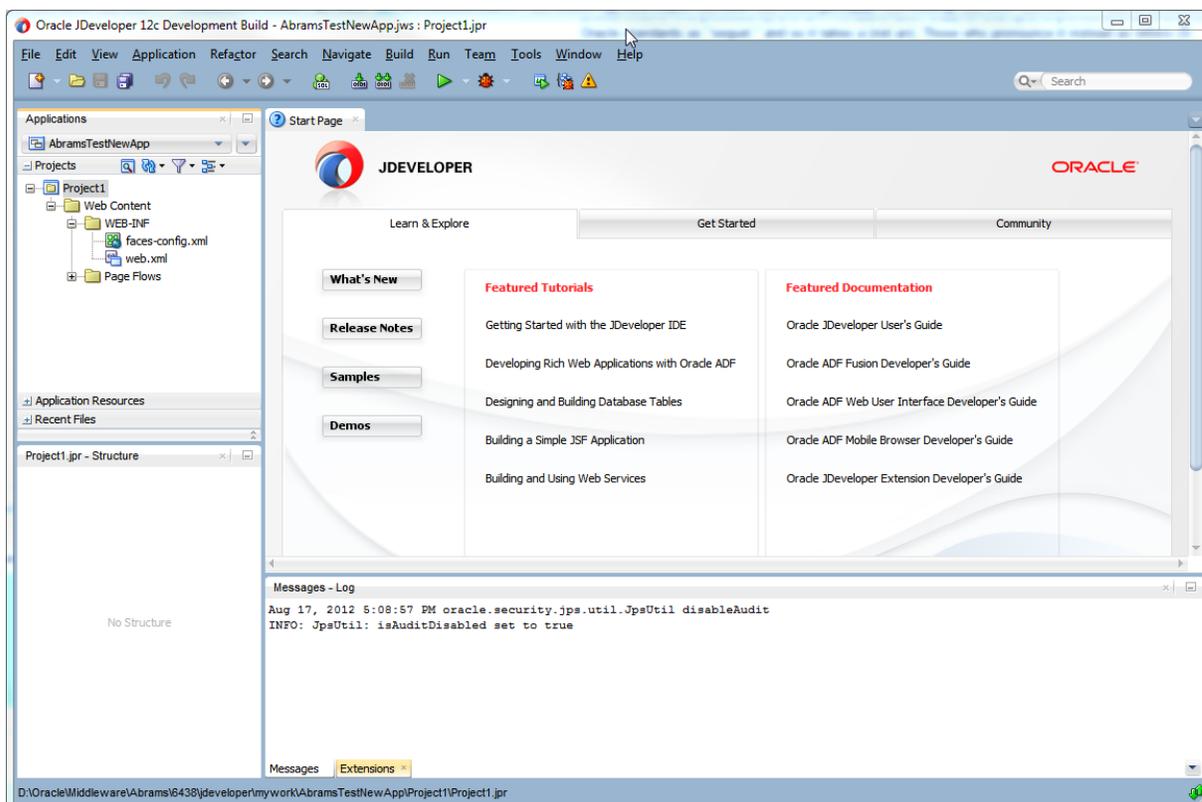
What Happens When You Create an Application Workspace

When you create an application workspace using the Custom template, and then select ADF Faces for your project, JDeveloper creates a project that contains all the source and configuration files needed for an ADF Faces application. Additionally, JDeveloper adds the following libraries to your project:

- JSF 2.2
- JSTL 1.2
- JSP Runtime

Once the projects are created for you, you can rename them. [Figure 3-1](#) shows the workspace for a new ADF Faces application.

Figure 3-1 New Workspace for an ADF Faces Application



JDeveloper also sets configuration parameters in the configuration files based on the options chosen when you created the application. In the `web.xml` file, these are configurations needed to run a JSF application (settings specific to ADF Faces are added when you create a JSF page with ADF Faces components). Following is the `web.xml` file generated by JDeveloper when you create a new ADF Faces application.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

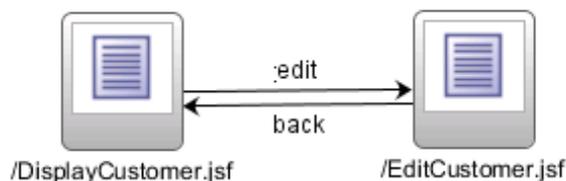
Configurations required for specific ADF Faces features are covered in the respective chapters of this guide. For example, any configuration needed in order to use the Change Persistence framework is covered in [Allowing User Customization on JSF Pages](#). For comprehensive information about configuring an ADF Faces application, see [ADF Faces Configuration](#).

Defining Page Flows

Defining the page flow of your ADF Faces application refers to how the UI elements are arranged in pages, designing the right sequence of pages, how pages interact with each other and how they can be modularized and communicate with each other as well. Navigation cases and rules are used to define the page flow.

Once you create your application workspace, often the next step is to design the flow of your UI. As with standard JSF applications, ADF Faces applications use navigation cases and rules to define the page flow. These definitions are stored in the `faces-config.xml` file. JDeveloper provides a diagrammer through which you can declaratively define your page flow using icons.

[Figure 3-2](#) shows the navigation diagram created for a simple page flow that contains two pages: a `DisplayCustomer` page that shows data for a specific customer, and an `EditCustomer` page that allows a user to edit the customer information. There is one navigation rule that goes from the display page to the edit page and one navigation rule that returns to the display page from the edit page.

Figure 3-2 Navigation Diagram in JDeveloper**Note:**

If you plan on using ADF Model data binding and ADF Controller, then you use ADF task flows to define your navigation rules. See “Getting Started with ADF Task Flows” in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Best Practice:

ADF Controller extends the JSF default controller. While you can technically use the JSF controller and ADF Controller in your application, you should use only one or the other.

With the advent of JSF 2.0, you no longer need to create a navigation case for simple navigation between two pages. If no matching navigation case is found after checking all available rules, the navigation handler checks to see whether the action outcome corresponds to a view ID. If a view matching the action outcome is found, an implicit navigation to the matching view occurs. For information on how navigation works in a JSF application, see the Java EE 6 tutorial (<http://download.oracle.com/javasee/index.html>).

How to Define a Page Flow

You use the navigation diagrammer to declaratively create a page flow using Facelets or JSPX pages. When you use the diagrammer, JDeveloper creates the XML metadata needed for navigation to work in your application in the `faces-config.xml` file.

Before you begin:

It may be helpful to have an understanding of page flows. For information, see [Defining Page Flows](#).

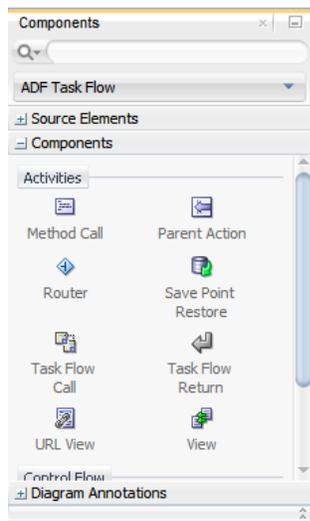
To create a page flow:

1. In the Applications window, double-click the `faces-config.xml` file for your application. By default, this is in the **Web Content/WEB-INF** node of your project.

2. In the editor window, click the **Diagram** tab to open the navigation diagrammer.
3. If the Components window is not displayed, from the main menu choose **Window > Components**. By default, the Components window is displayed in the upper right-hand corner of JDeveloper.
4. In the Components window, use the dropdown menu to choose **ADF Task Flow**.

The components are contained in three accordion panels: **Source Elements**, **Components**, and **Diagram Annotations**. [Figure 3-3](#) shows the Components window displaying JSF navigation components.

Figure 3-3 Components in JDeveloper



5. Select the component you wish to use and drag it onto the diagram. JDeveloper redraws the diagram with the newly added component.

 **Tip:**

You can also use the overview editor to create navigation rules and navigation cases by clicking the **Overview** tab. For help with the editor, click **Help** or press F1.

Additionally, you can manually add elements to the `faces-config.xml` file by directly editing the page in the source editor. To view the file in the source editor, click the **Source** tab.

Once the navigation for your application is defined, you can create the pages and add the components that will execute the navigation. For information about using navigation components on a page, see [Working with Navigation Components](#).

What Happens When You Use the Diagrammer to Create a Page Flow

When you use the diagrammer to create a page flow, JDeveloper creates the associated XML entries in the `faces-config.xml` file. This code shows the XML generated for the navigation rules displayed in [Figure 3-2](#).

```
<navigation-rule>
  <from-view-id>/DisplayCustomer</from-view-id>
  <navigation-case>
    <from-outcome>edit</from-outcome>
    <to-view-id>/EditCustomer</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/EditCustomer</from-view-id>
  <navigation-case>
    <from-outcome>back</from-outcome>
    <to-view-id>/DisplayCustomer</to-view-id>
  </navigation-case>
</navigation-rule>
```

Creating a View Page

The JSF pages you create for your ADF Faces application using JavaServer Faces can be Facelets documents (.jsf) or JSP documents written in XML syntax (.jspx). You can define the look and feel for the new page, and you can specify whether or not components on the page are exposed in a managed bean, to allow programmatic manipulation of the UI components.

From the page flows you created during the planning stages, you can double-click the page icons to create the actual JSF page files. You can choose to create either a Facelets page or a JSP page. Facelet pages use the extension *.jsf. Facelets is a JSF-centric declarative XML view definition technology that provides an alternative to using the JSP engine.

If instead you create a JSP page for an ADF Faces application, you create an XML-based JSP document, which uses the extension *.jspx. Using an XML-based document has the following advantages:

- It simplifies treating your page as a well-formed tree of UI component tags.
- It discourages you from mixing Java code and component tags.
- It allows you to easily parse the page to create documentation or audit reports.

Best Practice:

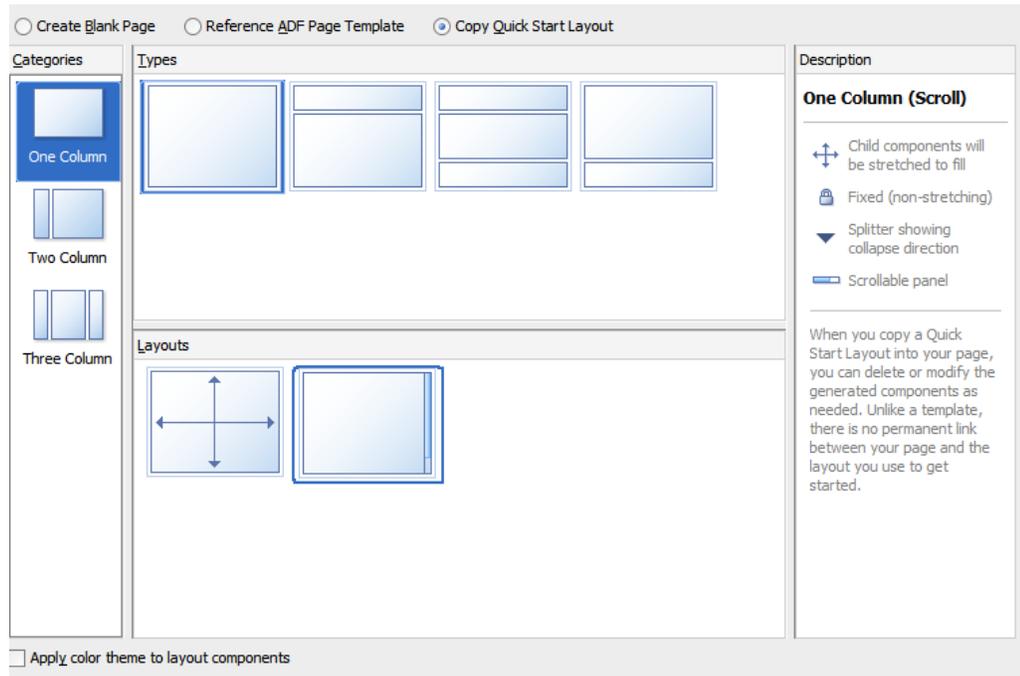
Use Facelets to take advantage of the following:

- The Facelets layer was created specifically for JSF, which results in reduced overhead and improved performance during tag compilation and execution.
- Facelets is considered the primary view definition technology in JSF 2.0.
- Some future performance enhancements to the JSF standard will only be available with Facelets.

ADF Faces provides a number of components that you can use to define the overall layout of a page. JDeveloper contains predefined quick start layouts that use these components to provide you with a quick and easy way to correctly build the layout. You can choose from one, two, or three column layouts, and then determine how you

want the columns to behave. For example, you may want one column's width to be locked, while another column stretches to fill available browser space. [Figure 3-4](#) shows the quick start layouts available for a two-column layout with the second column split between two panes. For information about the layout components, see [Organizing Content on Web Pages](#).

Figure 3-4 Quick Layouts



Best Practice:

Creating a layout that works correctly in all browsers can be time consuming. Use a predefined quick layout to avoid any potential issues.

Along with adding layout components, you can also choose to apply a theme to the chosen quick layout. These themes add color styling to some of the components used in the quick start layout. To see the color and where it is added, see [Quick Start Layout Themes](#). For information about themes, see [Customizing the Appearance Using Styles and Skins](#)

When you know you want to use the same layout on many pages in your application, ADF Faces allows you to create and use predefined page templates. When creating templates, the template developer can not only determine the layout of any page that will use the template, but can also provide static content that must appear on all pages, as well as create placeholder attributes that can be replaced with valid values for each individual page.

For example, ADF Faces ships with the Oracle Three-Column-Layout template. This template provides areas for specific content, such as branding, a header, and

copyright information, and also displays a static logo and busy icon, as shown in Figure 3-5.

Figure 3-5 Oracle Three Column Layout Template



Whenever a template is changed, for example if the layout changes, any page that uses the template will also be automatically updated. For information about creating and using templates, see [Using Page Templates](#).

Best Practice:

Use templates to ensure consistency and so that in the future, you can easily update multiple pages in an application.

At the time you create a JSF page, you can also choose to create an associated backing bean for the page. Backing beans allow you to access the components on the page programmatically. For information about using backing beans with JSF pages, see [What You May Need to Know About Automatic Component Binding](#).

Best Practice:

Create backing beans only for pages that contain components that must be accessed and manipulated programmatically. Use managed beans instead if you need only to provide additional functionality accessed through EL expressions on component attributes (such as listeners).

Once your page files are created, you can add UI components and work with the page source.

How to Create JSF Pages

You create JSF pages (either Facelets or JSP) using the Create JSF Page dialog.

Before you begin:

It may be helpful to have an understanding of the different options when creating a page. See [Creating a View Page](#).

To create a JSF page:

1. In the Applications window, right-click the node (directory) where you would like the page to be saved, and choose **New > Page**.

OR

From a navigation diagram, double-click a page icon for a page that has not yet been created.

2. Complete the Create JSF Page dialog. For help, click **Help** in the dialog. For information about the Managed Bean page, which can be used to automatically create a backing bean and associated bindings, see [What You May Need to Know About Automatic Component Binding](#).



Note:

While a Facelets page can use any extension you'd like, a Facelets page must use the `.jsf` extension to be customizable. See [Allowing User Customization on JSF Pages](#).

What Happens When You Create a JSF Page

When you use the Create JSF Page dialog to create a JSF page, JDeveloper creates the physical file and adds the code necessary to import the component libraries and display a page. The code created depends on whether or not you chose to create a Facelets or JSP page.

The following shows the code for a Facelets page when it is first created by JDeveloper.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html>
<f:view xmlns:f="http://java.sun.com/jsf/core" xmlns:af="http://xmlns.oracle.com/adf/
faces/rich">
  <af:document title="DisplayCustomer.jsf" id="dl">
    <af:form id="f1"></af:form>
  </af:document>
</f:view>
```

Below is the code for a `.jspx` page when it is first created by JDeveloper.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1" xmlns:f="http://
java.sun.com/jsf/core"
  xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
```

```

    <f:view>
      <af:document title="EditCustomer" id="d1">
        <af:form id="f1"></af:form>
      </af:document>
    </f:view>
  </jsp:root>

```

If you chose to use one of the quick layouts, then JDeveloper also adds the components necessary to display the layout. For example, the code below is what JDeveloper generates when you choose a two-column layout where the first column is locked and the second column stretches to fill up available browser space, and you also choose to apply themes.

```

<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1" xmlns:f="http://
java.sun.com/jsf/core"
  xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <f:view>
    <af:document title="EditCustomer" id="d1">
      <af:form id="f1">
        <af:panelGridLayout id="pgl1">
          <af:gridRow height="100%" id="gr1">
            <af:gridCell width="100px" valign="stretch"
              halign="stretch" id="gc1">
              <!-- Left -->
            </af:gridCell>
            <af:gridCell width="100%" valign="stretch"
              halign="stretch" id="gc2">
              <af:decorativeBox theme="dark" id="db2">
                <f:facet name="center">
                  <af:decorativeBox theme="medium" id="db1">
                    <f:facet name="center">
                      <!-- Content -->
                    </f:facet>
                  </af:decorativeBox>
                </f:facet>
              </af:decorativeBox>
            </af:gridCell>
          </af:gridRow>
        </af:panelGridLayout>
      </af:form>
    </af:document>
  </f:view>
</jsp:root>

```

If you chose to automatically create a backing bean using the Managed Bean tab of the dialog, JDeveloper also creates and registers a backing bean for the page, and binds any existing components to the bean, as shown in the code below.

```

package view.backing;

import oracle.adf.view.rich.component.rich.RichDocument;
import oracle.adf.view.rich.component.rich.RichForm;

public class MyFile {
  private RichForm f1;
  private RichDocument dl;

  public void setF1(RichForm f1) {
    this.f1 = f1;
  }
}

```

```
    }  
  
    public RichForm getF1() {  
        return f1;  
    }  
  
    public void setD1(RichDocument d1) {  
        this.document1 = d1;  
    }  
  
    public RichDocument getD1() {  
        return d1;  
    }  
}
```

 **Tip:**

You can access the backing bean source from the JSF page by right-clicking the page in the editor, and choosing **Go to** and then selecting the bean from the list.

Additionally, JDeveloper adds the following libraries to the view project:

- ADF Faces Runtime 11
- ADF Common Runtime
- ADF DVT Faces Runtime
- ADF DVT Faces Databinding Runtime
- ADF DVT Faces Databinding MDS Runtime
- Oracle JEWTT

JDeveloper also adds entries to the `web.xml` file. Following is the `web.xml` file created once you create a JSF page.

```
<?xml version = '1.0' encoding = 'windows-1252'?>  
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/  
XMLSchema-instance"  
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"  
        version="3.0">  
    <servlet>  
        <servlet-name>Faces Servlet</servlet-name>  
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>  
        <load-on-startup>1</load-on-startup>  
    </servlet>  
    <servlet>  
        <servlet-name>resources</servlet-name>  
        <servlet-class>  
            org.apache.myfaces.trinidad.webapp.ResourceServlet  
        </servlet-class>  
    </servlet>  
    <servlet>  
        <servlet-name>BIGRAPHSERVLET</servlet-name>  
        <servlet-class>oracle.adf.view.faces.bi.webapp.GraphServlet</servlet-class>  
    </servlet>  
</web-app>
```

```

    <servlet-name>BIGAUGESERVLET</servlet-name>
    <servlet-class>oracle.adf.view.faces.bi.webapp.GaugeServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>MapProxyServlet</servlet-name>
    <servlet-class>oracle.adf.view.faces.bi.webapp.MapProxyServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/adf/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/afr/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>BIGRAPHSERVLET</servlet-name>
    <url-pattern>/servlet/GraphServlet/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>BIGAUGESERVLET</servlet-name>
    <url-pattern>/servlet/GaugeServlet/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>MapProxyServlet</servlet-name>
    <url-pattern>/mapproxy/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/bi/*</url-pattern>
</servlet-mapping>
<context-param>
    <param-name>javax.faces.FACELETS_VIEW_MAPPINGS</param-name>
    <param-value>*.jsf;*.xhtml</param-value>
</context-param>
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>
<context-param>
    <param-name>javax.faces.PARTIAL_STATE_SAVING</param-name>
    <param-value>>false</param-value>
</context-param>
<context-param>
    <description>If this parameter is true, there will be an automatic check of the
modification date of your JSPs, and saved state will be discarded when JSP's change.
It will also automatically check if your skinning css files have changed without you
having to restart the server. This makes development easier, but adds overhead. For
this reason this parameter should be set to false when your application is
deployed.</description>
    <param-name>org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION</param-name>
    <param-value>>false</param-value>
</context-param>
<context-param>
    <description>Whether the 'Generated by...' comment at the bottom of ADF Faces
HTML pages should contain version number information.</description>
    <param-name>oracle.adf.view.rich.versionString.HIDDEN</param-name>

```

```
<param-value>>false</param-value>
</context-param>
<context-param>
  <description>Security precaution to prevent clickjacking: bust frames if the
  ancestor window domain(protocol, host, and port) and the frame domain are different.
  Another options for this parameter are always and never.</description>
  <param-name>org.apache.myfaces.trinidad.security.FRAME_BUSTING</param-name>
  <param-value>differentOrigin</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.VALIDATE_EMPTY_FIELDS</param-name>
  <param-value>>true</param-value>
</context-param>
<context-param>
  <param-name>oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS</param-name>
  <param-value>auto</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.FACELETS_SKIP_XML_INSTRUCTIONS</param-name>
  <param-value>>true</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
  <param-value>>true</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.FACELETS_DECORATORS</param-name>
  <param-value>
    oracle.adfinternal.view.faces.facelets.rich.AdfTagDecorator
  </param-value>
</context-param>
<context-param>
  <param-name>javax.faces.FACELETS_RESOURCE_RESOLVER</param-name>
  <param-value>
    oracle.adfinternal.view.faces.facelets.rich.AdfFaceletsResourceResolver
  </param-value>
</context-param>
<context-param>
  <param-name>oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT</param-name>
  <param-value>HTML5</param-value>
</context-param>
<filter>
  <filter-name>trinidad</filter-name>
  <filter-class>org.apache.myfaces.trinidad.webapp.TrinidadFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>trinidad</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
<mime-mapping>
  <extension>swf</extension>
  <mime-type>application/x-shockwave-flash</mime-type>
</mime-mapping>
<mime-mapping>
  <extension>amf</extension>
  <mime-type>application/x-amf</mime-type>
</mime-mapping>
</web-app>
```

 **Note:**

The Facelets context parameters are only created if you create a Facelets page.

In the `faces-config.xml` file, when you create a JSF page, JDeveloper creates an entry that defines the default render kit (used to display the components in an HTML client) for ADF Faces, as shown below.

```
<?xml version="1.0" encoding="windows-1252"?>
<faces-config version="2.1" xmlns="http://java.sun.com/xml/ns/javaee">
  <application>
    <default-render-kit-id>oracle.adf.rich</default-render-kit-id>
  </application>
</faces-config>
```

An entry in the `trinidad-config.xml` file defines the default skin used by the user interface (UI) components in the application, as shown in below.

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>skyros</skin-family>
    <skin-version>v1</skin-version>
</trinidad-config>
```

When the page is first displayed in JDeveloper, it is displayed in the visual editor (accessed by clicking the **Design** tab), which allows you to view the page in a WYSIWYG environment. You can also view the source for the page in the source editor by clicking the **Source** tab. The Structure window located in the lower left-hand corner of JDeveloper, provides a hierarchical view of the page.

What You May Need to Know About Updating Your Application to Use the Facelets Engine

JSF 2.1 web applications can run using either the Facelets engine or JSP servlet engine. By default, documents with the `*.jsf` and `*.xhtml` extensions are handled by the Facelets engine, while documents with the `*.jsp` and `*.jspx` extensions are handled by the JSP engine. However, this behavior may be changed by setting the `javax.faces.FACELETS_VIEW_MAPPINGS` context parameter in the `web.xml` file. Because ADF Faces allows JSP pages to be run with the Facelets engine, you may decide that you want an existing application of JSP pages to use the Facelets engine. To do that, insert the following code into your `web.xml` page.

```
<context-param>
  <param-name>javax.faces.FACELETS_VIEW_MAPPINGS</param-name>
  <!-- Map both *.jspx and *.jsf to the Facelets engine -->
  <param-value>*.jsf; *.jspx</param-value>
</context-param>

<context-param>
  <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
  <param-value>>true</param-value>
</context-param>

<context-param>
```

```
<param-name>javax.faces.FACELETS_DECORATORS</param-name>
<param-value>
  oracle.adfinternal.view.faces.facelets.rich.AdfTagDecorator
</param-value>
</context-param>

<context-param>
<param-name>
  javax.faces.FACELETS_RESOURCE_RESOLVER
</param-name>
<param-value>
  oracle.adfinternal.view.faces.facelets.rich.AdfFaceletsResourceResolver
</param-value>
</context-param>
```

You then must redeploy your ADF Faces libraries.

Note that if you do change your application to use the Facelets engine, then your application will use JSF partial state saving, which is not currently compatible with ADF Faces. You will need to explicitly add the entry shown below.

```
<context-param>
  <param-name>javax.faces.PARTIAL_STATE_SAVING</param-name>
  <param-value>>false</param-value>
</context-param>
```

Once this incompatibility is resolved, you should re-enable partial state saving by removing the entry. Check your current release notes at <http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html> for the latest information on partial state saving support.

 **Note:**

When you switch from the servlet engine to the Facelets engine, you may find certain parts of your application do not function as expected. For example, if you have any custom JSP tags, these tags will need to be reimplemented to work with the Facelets engine. For more information, refer to the ADF Faces release notes.

What You May Need to Know About Automatic Component Binding

Backing beans are managed beans that contain logic and properties for UI components on a JSF page (for information about managed beans, see [Creating and Using Managed Beans](#)). If when you create your JSF page you choose to automatically expose UI components by selecting one of the choices in the Page Implementation option of the Create JSF Page dialog, JDeveloper automatically creates a backing bean (or uses a managed bean of your choice) for the page. For each component you add to the page, JDeveloper then inserts a bean property for that component, and uses the `binding` attribute to bind component instances to those properties, allowing the bean to accept and return component instances.

Specifically, JDeveloper does the following when you use automatic component binding:

- Creates a JavaBean using the same name as the JSF file, and places it in the `view.backing` package (if you elect to have JDeveloper create a backing bean).
- Creates a managed bean entry in the `faces-config.xml` file for the backing bean. By default, the managed bean name is `backing_<page_name>` and the bean uses the `request` scope (for information about scopes, see [Object Scope Lifecycles](#)).

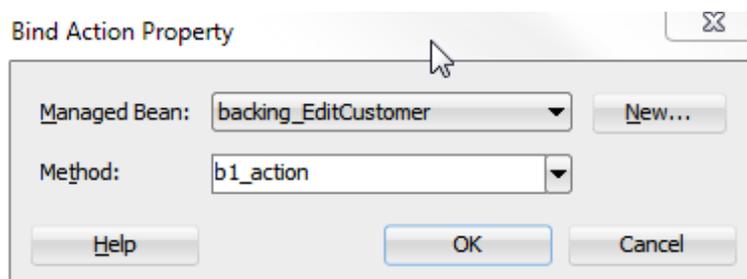
 **Note:**

JDeveloper does not create managed bean property entries in the `faces-config.xml` file. If you wish the bean to be instantiated with certain property values, you must perform this configuration in the `faces-config.xml` file manually. See [How to Configure for ADF Faces in faces-config.xml](#).

- On the newly created or selected bean, adds a property and accessor methods for each component tag you place on the JSF page. JDeveloper binds the component tag to that property using an EL expression as the value for its binding attribute.
- Deletes properties and methods for any components deleted from the page.

Once the page is created and components added, you can then declaratively add method binding expressions to components that use them by double-clicking the component in the visual editor, which launches an editor that allows you to select the managed bean and method to which you want to bind the attribute. When automatic component binding is used on a page and you double-click the component, skeleton methods to which the component may be bound are automatically created for you in the page's backing bean. For example, if you add a button component and then double-click it in the visual editor, the Bind Action Property dialog displays the page's backing bean along with a new skeleton action method, as shown in [Figure 3-6](#).

Figure 3-6 Bind Action Property Dialog



You can select from one these methods, or if you enter a new method name, JDeveloper automatically creates the new skeleton method in the page's backing bean. You must then add the logic to the method.

 **Note:**

When automatic component binding is *not* used on a page, you must select an existing managed bean or create a new backing bean to create the binding.

For example, suppose you created a JSF page with the file name `myfile.jsf`. If you chose to let JDeveloper automatically create a default backing bean, then JDeveloper creates the backing bean as `view.backing.MyFile.java`, and places it in the `\src` directory of the `ViewController` project. The backing bean is configured as a managed bean in the `faces-config.xml` file, and the default managed bean name is `backing_myfile`.

The following code from a JSF page uses automatic component binding, and contains form, `inputText`, and `button` components.

```
<f:view>
  <af:document id="d1" binding="#{backing_myfile.d1}">
    <af:form id="f1" binding="#{backing_myfile.f1}">
      <af:inputText label="Label 1" binding="#{backing_MyFile.it1}"
        id="inputText1"/>
      <af:button text="button 1"
        binding="#{backing_MyFile.b1}"
        id="b1"/>
    </af:form>
  </af:document>
</f:view>
```

Following is the corresponding code on the backing bean.

```
package view.backing;

import oracle.adf.view.rich.component.rich.RichDocument;
import oracle.adf.view.rich.component.rich.RichForm;
import oracle.adf.view.rich.component.rich.input.RichInputText;
import oracle.adf.view.rich.component.rich.nav.RichButton;

public class MyFile {
    private RichForm f1;
    private RichDocument d1;
    private RichInputText it1;
    private RichButton b1;

    public void setForm1(RichForm f1) {
        this.form1 = f1;
    }

    public RichForm getF1() {
        return f1;
    }

    public void setD1(RichDocument d1) {
        this.d1 = d1;
    }

    public RichDocument getD1() {
        return d1;
    }
}
```

```
    }

    public void setIt1(RichInputText it1) {
        this.inputText1 = inputText1;
    }

    public RichInputText getInputText1() {
        return inputText1;
    }

    public void setB1(RichButton b1) {
        this.button1 = button1;
    }

    public RichButton getB1() {
        return b1;
    }

    public String b1_action() {
        // Add event code here...
        return null;
    }
}
```

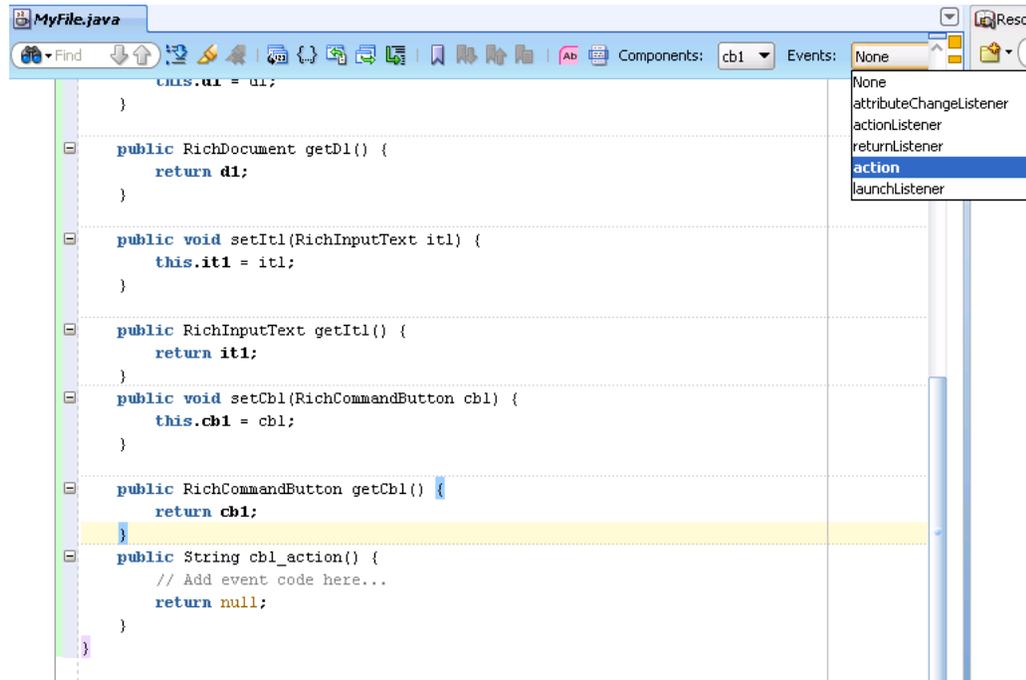
This code, added to the `faces-config.xml` file, registers the page's backing bean as a managed bean.

```
<managed-bean>
  <managed-bean-name>backing_MyFile</managed-bean-name>
  <managed-bean-class>view.backing.MyFile</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

 **Note:**

Instead of registering the managed bean in the `faces-config.xml` file, if you are using Facelets, you can elect to use annotations in the backing bean for registration. For information about using annotations in managed and backing beans, see the Java EE 6 tutorial at <http://www.oracle.com/technetwork/java/index.html>.

In addition, when you edit a Java file that is a backing bean for a JSF page, a method binding toolbar appears in the source editor for you to bind appropriate methods quickly and easily to selected components in the page. When you select an event, JDeveloper creates the skeleton method for the event, as shown in [Figure 3-7](#).

Figure 3-7 You Can Declaratively Create Skeleton Methods in the Source Editor

Once you create a page, you can turn automatic component binding off or on, and you can also change the backing bean to a different Java class. Open the JSF page in the visual Editor and from the JDeveloper menu, choose **Design > Page Properties**. Here you can select or deselect the **Auto Bind** option, and change the managed bean class. Click **Help** for information about using the dialog.

Note:

If you turn automatic binding off, nothing changes in the binding attributes of existing bound components in the page. If you turn automatic binding on, all existing bound components and any new components that you insert are bound to the selected backing bean. If automatic binding is on and you change the bean selection, all existing bindings and new bindings are switched to the new bean.

You can always access the backing bean for a JSF page from the page editor by right-clicking the page, choosing **Go to Bean**, and then choosing the bean from the list of beans associated with the JSF.

How to Add ADF Faces Components to JSF Pages

Once you have created a page, you can use the Components window to drag and drop components onto the page. JDeveloper then declaratively adds the necessary page code and sets certain values for component attributes.

 **Tip:**

For detailed procedures and information about adding and using specific ADF Faces components, see [Using Common ADF Faces Components](#).

 **Note:**

You cannot use ADF Faces components on the same page as MyFaces Trinidad components (`tr:` tags) or other Ajax-enabled library components. You can use Trinidad HTML tags (`trh:`) on the same page as ADF Faces components, however you may experience some browser layout issues. You should always attempt to use only ADF Faces components to achieve your layout.

Note that your application may contain a mix of pages built using either ADF Faces or other components.

Before you begin:

It may be helpful to have an understanding of creating a page. See [Creating a View Page](#).

To add ADF Faces components to a page:

1. In the Applications window, double-click a JSF page to open it.
2. If the Components window is not displayed, from the menu choose **Window > Components**. By default, the Components window is displayed in the upper right-hand corner of JDeveloper.
3. In the Components window, use the dropdown menu to choose **ADF Faces**.

 **Tip:**

If the ADF Faces page is not available in the Components window, then you need to add the ADF Faces tag library to the project.

For a Facelets file:

- a. Right-click the project node and choose **Project Properties**.
- b. Select **JSP Tag Libraries** to add the ADF Faces library to the project. For help, click **Help** or press F1.

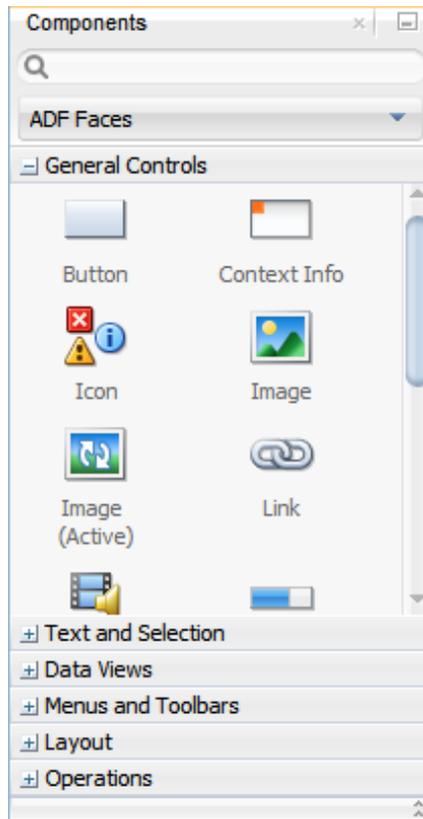
For a JSPX file:

- a. Right-click inside the Components window and choose **Edit Tab Libraries**.
- b. In the Customize Components window dialog, shuttle **ADF Faces Components** to **Selected Libraries**, and click **OK**.

The components are contained in 6 accordion panels: **General Controls** (which contains components like buttons, icons, and menus), **Text and Selection**, **Data Views** (which contains components like tables and trees), **Menus and Toolbars**, **Layout**, and **Operations**.

Figure 3-8 shows the Components Window displaying the general controls for ADF Faces.

Figure 3-8 Components Window in JDeveloper



4. Select the component you wish to use and drag it onto the page.

JDeveloper redraws the page in the visual editor with the newly added component. In the visual editor, you can directly select components on the page and use the resulting context menu to add more components.

 **Tip:**

You can also drag and drop components from the Components window into the Structure window or directly into the code in the source editor.

You can always add components by directly editing the page in the source editor. To view the page in the source editor, click the **Source** tab at the bottom of the window.

What Happens When You Add Components to a Page

When you drag and drop components from the Components window onto a JSF page, JDeveloper adds the corresponding code to the JSF page. This code includes the tag necessary to render the component, as well as values for some of the component attributes. For example, the following shows the code when you drop an Input Text and a Button component from the palette.

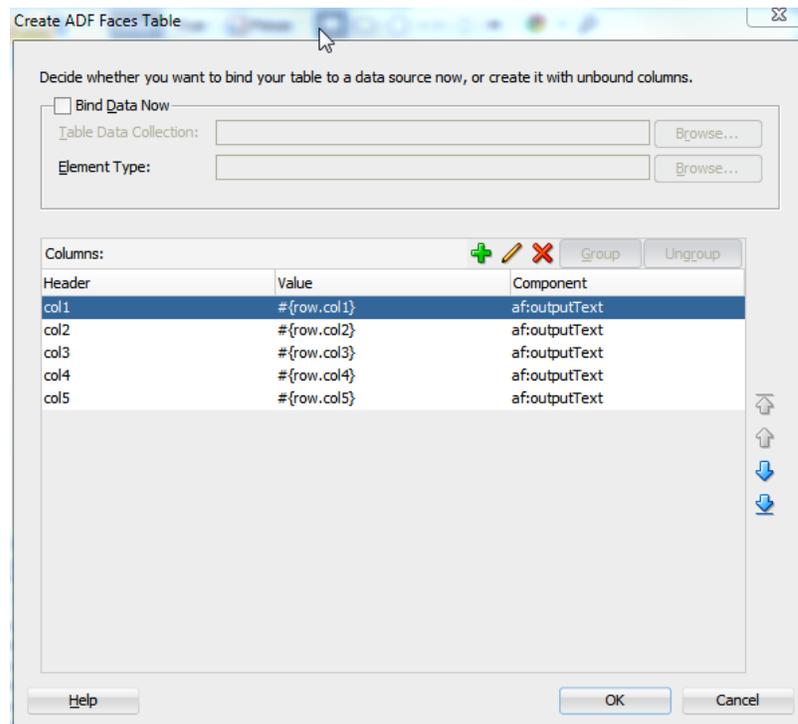
```
<af:inputText label="Label 1" id="it1"/>  
<af:button text="button 1" id="b1"/>
```

Note:

If you chose to use automatic component binding, then JDeveloper also adds the `binding` attribute with its value bound to the corresponding property on the page's backing bean. See [What You May Need to Know About Automatic Component Binding](#).

When you drop a component that contains mandatory child components (for example a table or a list), JDeveloper launches a wizard where you define the parent and also each of the child components. [Figure 3-9](#) shows the Table wizard used to create a table component and the table's child column components.

Figure 3-9 Table Wizard in JDeveloper



Below is the code created when you use the wizard to create a table with three columns, each of which uses an `outputText` component to display data.

```
<af:table var="row" rowBandingInterval="0" id="t1">
  <af:column sortable="false" headerText="col1" id="c1">
    <af:outputText value="#{row.col1}" id="ot1"/>
  </af:column>
  <af:column sortable="false" headerText="col2" id="c2">
    <af:outputText value="#{row.col2}" id="ot2"/>
  </af:column>
  <af:column sortable="false" headerText="col3" id="c3">
    <af:outputText value="#{row.col3}" id="ot3"/>
  </af:column>
</af:table>
```

How to Set Component Attributes

Once you drop components onto a page you can use the Properties window (displayed by default at the bottom right of JDeveloper) to set attribute values for each component.

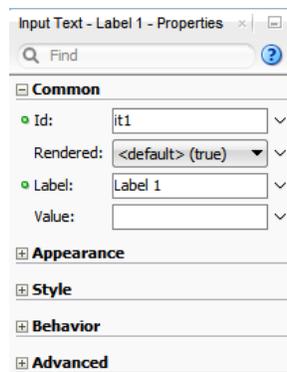


Tip:

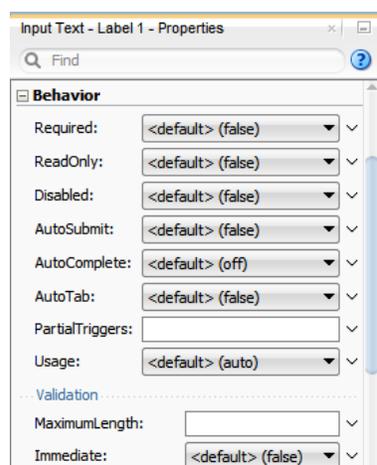
If the Properties window is not displayed, choose **Window > Properties** from the main menu.

Figure 3-10 shows the Properties window displaying the attributes for an `inputText` component.

Figure 3-10 JDeveloper Properties Window



The Properties window has sections that group similar properties together. For example, the Properties window groups commonly used attributes for the `inputText` component in the **Common** section, while properties that affect how the component behaves are grouped together in the **Behavior** section. Figure 3-11 shows the Behavior section of the Properties window for an `inputText` component.

Figure 3-11 Behavior Section of the Properties window

Before you begin:

It may be helpful to have an understanding of the different options when creating a page. For information, see [Creating a View Page](#).

To set component attributes:

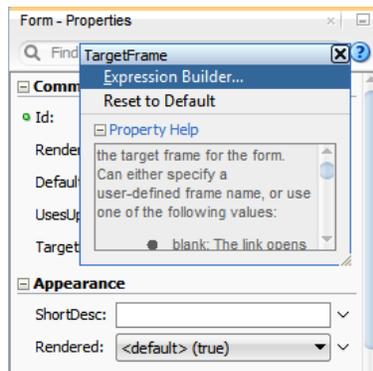
1. Select the component, in the visual editor, in the Structure window, or by selecting the tag directly in the source editor.
2. In the Properties window, expand the section that contains the attribute you wish to set.

 **Tip:**

Some attributes are displayed in more than one section. Entering or changing the value in one section will also change it in any other sections. You can search for an attribute by entering the attribute name in the search field at the top of the inspector.

3. Either enter values directly into the fields, or if the field contains a dropdown list, use that list to select a value. You can also use the dropdown to the right of the field, which launches a popup containing tools you can use to set the value. These tools are either specific property editors (opened by choosing **Edit**) or the Expression Builder, which you can use to create EL expressions for the value (opened by choosing **Expression Builder**). For information about using the Expression Builder, see [Creating EL Expressions](#). This popup also displays a description of the property, as shown in [Figure 3-12](#).

Figure 3-12 Property Tools and Help



What Happens When You Use the Properties window

When you use the Properties window to set or change attribute values, JDeveloper automatically changes the page source for the attribute to match the entered value.

Tip:

You can always change attribute values by directly editing the page in the source editor. To view the page in the source editor, click the **Source** tab at the bottom of the window.

Creating EL Expressions

In ADF applications, Express Language (EL) expressions provide an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans). The EL expressions are used by both JavaServer Faces technology (JSF) and JavaServer Pages (JSP) technology.

You use EL expressions throughout an ADF Faces application to bind attributes to object values determined at runtime. For example, `#{UserList.selectedUsers}` might reference a set of selected users, `#{user.name}` might reference a particular user's name, while `#{user.role == 'manager'}` would evaluate whether a user is a manager or not. At runtime, a generic expression evaluator returns the `List`, `String`, and `boolean` values of these respective expressions, automating access to the individual objects and their properties without requiring code.

At runtime, the value of certain JSF UI components (such as an `inputText` component or an `outputText` component) is determined by its `value` attribute. While a component can have static text as its value, typically the `value` attribute will contain an EL expression that the runtime infrastructure evaluates to determine what data to display. For example, an `outputText` component that displays the name of the currently logged-in user might have its `value` attribute set to the expression `#{UserInfo.name}`. Since any attribute of a component (and not just the `value` attribute) can be assigned a value using an EL expression, it's easy to build dynamic, data-driven user interfaces. For example, you could hide a component when a set of objects you need to display is empty by using a boolean-valued expression like `#{not empty userList.selectedUsers}` in the UI component's `rendered` attribute. If the list of

selected users in the object named `UserList` is empty, the `rendered` attribute evaluates to `false` and the component disappears from the page.

In a typical JSF application, you would create objects like `UserList` as a managed bean. The JSF runtime manages instantiating these beans on demand when any EL expression references them for the first time. When displaying a value, the runtime evaluates the EL expression and pulls the value from the managed bean to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF runtime pushes the value back into the corresponding managed bean based on the same EL expression. For information about creating and using managed beans, see [Creating and Using Managed Beans](#). For information about EL expressions, see the Java EE 6 tutorial at <http://www.oracle.com/technetwork/java/index.html>.

 **Note:**

When using an EL expression for the `value` attribute of an editable component, you must have a corresponding set method for the that component, or else the EL expression will evaluate to read-only, and no updates to the value will be allowed.

For example, say you have an `inputText` component (whose ID is `it1`) on a page, and you have its value set to `#{myBean.inputValue}`. The `myBean` managed bean would have to have get and set method as follows, in order for the `inputText` value to be updated:

```
public void setIt1(RichInputText it1) {
    this.it1 = it1;
}

public RichInputText getIt1() {
    return it1;
}
```

Along with standard EL reachable objects and operands, ADF Faces provides EL function tags. These are tags that provide certain functionality that you can use within an EL expression. The format tags can be used to add parameters to String messages, and the time zone tags can be used to return time zones. For information about the format tags, see [How to Use the EL Format Tags](#). For information about the time zone tags, see [What You May Need to Know About Selecting Time Zones Without the inputDate Component](#).

How to Create an EL Expression

You can create EL expressions declaratively using the JDeveloper Expression Builder. You can access the builder from the Properties window.

Before you begin:

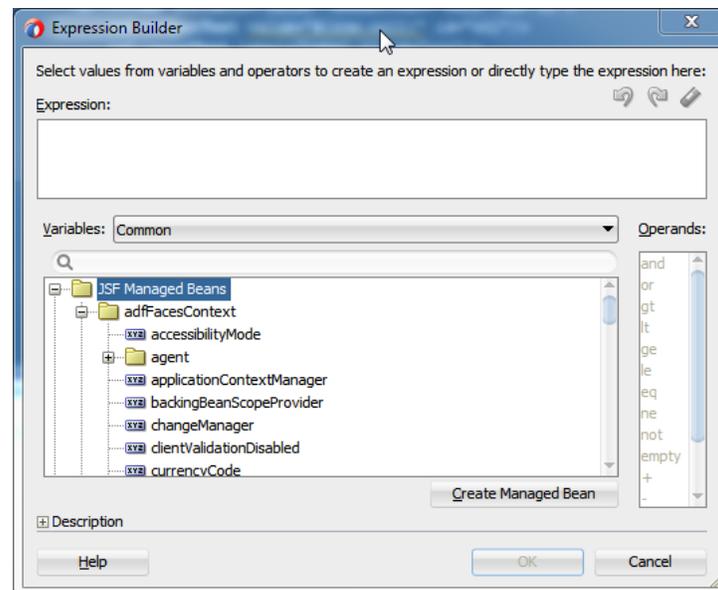
It may be helpful to have an understanding of EL expressions. See [Creating EL Expressions](#).

To use the Expression Builder:

1. In the Properties window, locate the attribute you wish to modify and use the rightmost dropdown menu to choose **Expression Builder**.
2. Create expressions using the following features:
 - Use the **Variables** dropdown to select items that you want to include in the expression. These items are displayed in a tree that is a hierarchical representation of the binding objects. Each icon in the tree represents various types of binding objects that you can use in an expression.

To narrow down the tree, you can either use the dropdown filter or enter search criteria in the search field. The EL accessible objects exposed by ADF Faces are located under the **adfFacesContext** node, which is under the **JSF Managed Beans** node, as shown in [Figure 3-13](#).

Figure 3-13 adfFacesContext Objects in the Expression Builder



 **Tip:**

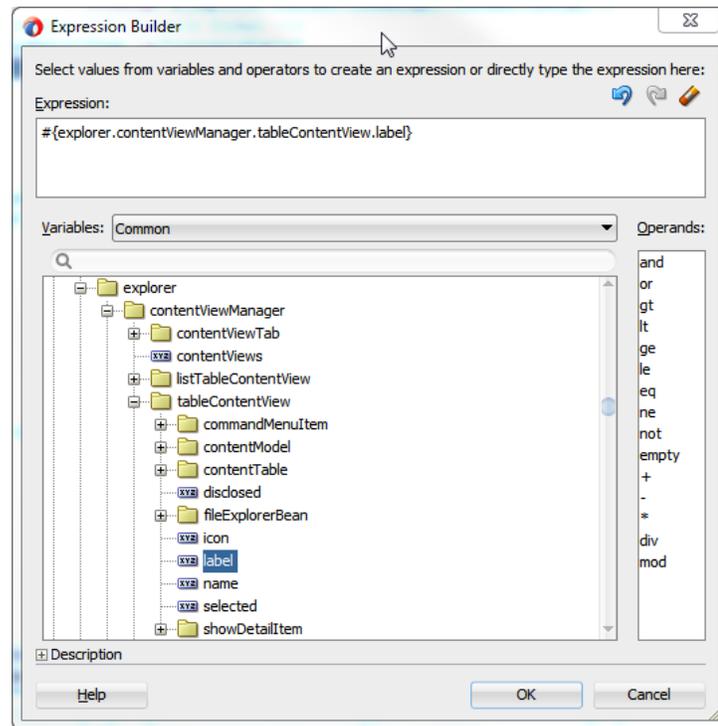
For information about these objects, see the *Java API Reference for Oracle ADF Faces*.

Selecting an item in the tree causes it to be moved to the **Expression** box within an EL expression. You can also type the expression directly in the **Expression** box.

- Use the operator buttons to add logical or mathematical operators to the expression.

[Figure 3-14](#) shows the Expression Builder dialog being used to create an expression that binds to the value of a label for a component to the `label` property of the `explorer` managed bean.

Figure 3-14 The Expression Builder Dialog

 **Tip:**

For information about using proper syntax to create EL expressions, see the Java EE 6 tutorial at <http://download.oracle.com/javaee/index.html>.

How to Use the EL Format Tags

ADF EL format tags allow you to create text that uses placeholder parameters, which can then be used as the value for any component attribute that accepts a *String*. At runtime, the placeholders are replaced with the parameter values.

For example, say the current user's name is stored on a managed bean, and you want to display that name within a message as the value of an *outputText* component. You could use the *formatString* tag as shown below.

```
<af:outputText value="#{af:formatString('The current user is: {0},  
someBean.currentUser)}" />
```

In this example, the *formatString* tag takes one parameter whose key "0," resolves to the value *someBean.currentUser*.

There are two different types of format tags available, *formatString* tags and *formatNamed* tags. The *formatString* tags use indexed parameters, while the *formatNamed* tags use named parameters. There are four tags for each type, each one taking a different number of parameters (up to 4). For example, the *formatString2* tag

takes two indexed parameters, and the `formatNamed4` tag takes four named parameters.

When you use a `formatNamed` tag, you set both the key and the value. The following example shows a message that uses the `formatNamed2` tag to display the number of files on a specific disk. This message contains two parameters.

```
<af:outputText value="#{af:formatNamed2(
  'The disk named {disk}, contains {fileNumber} files', 'disk', bean.disk,
  'fileNumber', bean.fileNumber)}" />
```

How to Use EL Expressions Within Managed Beans

While JDeveloper creates many needed EL expressions for you, and you can use the Expression Builder to create those not built for you, there may be times when you need to access, set, or invoke EL expressions within a managed bean.

The code below shows how you can get a reference to an EL expression and return (or create) the matching object.

```
public static Object resolveExpression(String expression) {
    FacesContext facesContext = getFacesContext();
    Application app = facesContext.getApplication();
    ExpressionFactory elFactory = app.getExpressionFactory();
    ELContext elContext = facesContext.getELContext();
    ValueExpression valueExp =
        elFactory.createValueExpression(elContext, expression,
                                       Object.class);
    return valueExp.getValue(elContext);
}
```

This code shows how you can resolve a method expression.

```
public static Object resolveMethodExpression(String expression,
                                           Class returnType,
                                           Class[] argTypes,
                                           Object[] argValues) {
    FacesContext facesContext = getFacesContext();
    Application app = facesContext.getApplication();
    ExpressionFactory elFactory = app.getExpressionFactory();
    ELContext elContext = facesContext.getELContext();
    MethodExpression methodExpression =
        elFactory.createMethodExpression(elContext, expression, returnType,
                                       argTypes);
    return methodExpression.invoke(elContext, argValues);
}
```

The code below shows how you can set a new object on a managed bean.

```
public static void setObject(String expression, Object newValue) {
    FacesContext facesContext = getFacesContext();
    Application app = facesContext.getApplication();
    ExpressionFactory elFactory = app.getExpressionFactory();
    ELContext elContext = facesContext.getELContext();
    ValueExpression valueExp =
        elFactory.createValueExpression(elContext, expression,
                                       Object.class);

    //Check that the input newValue can be cast to the property type
    //expected by the managed bean.
```

```
//Rely on Auto-Unboxing if the managed Bean expects a primitive
Class bindClass = valueExp.getType(elContext);
if (bindClass.isPrimitive() || bindClass.isInstance(newValue)) {
    valueExp.setValue(elContext, newValue);
}
}
```

Creating and Using Managed Beans

A managed bean is created with a constructor with no arguments, a set of properties, and a set of methods that perform functions for a component. In the ADF framework, if you want to bind component values and objects to managed bean properties or to reference managed bean methods from component tags, you can use the EL syntax.

Managed beans are Java classes that you register with the application using various configuration files. When the JSF application starts up, it parses these configuration files and the beans are made available and can be referenced in an EL expression, allowing access to the beans' properties and methods. Whenever a managed bean is referenced for the first time and it does not already exist, the Managed Bean Creation Facility instantiates the bean by calling the default constructor method on the bean. If any properties are also declared, they are populated with the declared default values.

Often, managed beans handle events or some manipulation of data that is best handled at the front end. For a more complete description of how managed beans are used in a standard JSF application, see the Java EE 6 tutorial at <http://www.oracle.com/technetwork/java/index.html>.

Best Practice:

Use managed beans to store only bookkeeping information, for example the current user. All application data and processing should be handled by logic in the business layer of the application.

In a standard JSF application, managed beans are registered in the `faces-config.xml` configuration file.

Note:

If you plan on using ADF Model data binding and ADF Controller, then instead of registering managed beans in the `faces-config.xml` file, you may need to register them within ADF task flows. See *Using a Managed Bean in a Fusion Web Application* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

How to Create a Managed Bean in JDeveloper

You can create a managed bean and register it with the JSF application at the same time using the overview editor for the `faces-config.xml` file.

Before you begin:

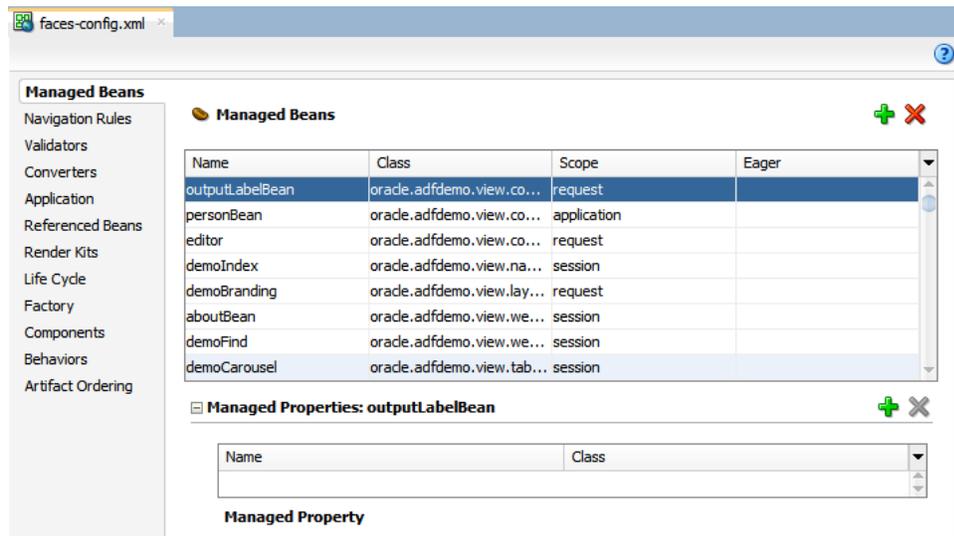
It may be helpful to have an understanding of managed beans. See [Creating and Using Managed Beans](#).

To create and register a managed bean:

1. In the Applications window, double-click **faces-config.xml**.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Managed Beans** navigation tab.

[Figure 3-15](#) shows the editor for the `faces-config.xml` file used by the ADF Faces Components Demo application that contains the File Explorer application.

Figure 3-15 Managed Beans in the `faces-config.xml` File



4. Click the **Add** icon to add a row to the Managed Bean table.
5. In the Create Managed Bean dialog, enter values. Click **Help** for information about using the dialog. Select the **Generate Class If It Does Not Exist** option if you want JDeveloper to create the class file for you.

 **Note:**

When determining what scope to register a managed bean with or to store a value in, keep the following in mind:

- Always try to use the narrowest scope possible.
- If your managed bean takes part in component binding by accepting and returning component instances (that is, if UI components on the page use the `binding` attribute to bind to component properties on the bean), then the managed bean must be stored in `request` or `backingBean` scope. If it can't be stored in one of those scopes (for example, if it needs to be stored in `session` scope for high availability reasons), then you need to use the `ComponentReference` API. See [What You May Need to Know About Component Bindings and Managed Beans](#).
- Use the `sessionScope` scope only for information that is relevant to the whole session, such as user or context information. Avoid using the `sessionScope` scope to pass values from one page to another.

For information about the different object scopes, see [Object Scope Lifecycles](#).

6. You can optionally add managed properties for the bean. When the bean is instantiated, any managed properties will be set with the provided value. With the bean selected in the Managed Bean table, click the **New** icon to add a row to the Managed Properties table. In the Properties window, enter a property name (other fields are optional).

 **Note:**

While you can declare managed properties using this editor, the corresponding code is not generated on the Java class. You must add that code by creating private member fields of the appropriate type, and then by choosing the **Generate Accessors** menu item on the context menu of the code editor to generate the corresponding `get` and `set` methods for these bean properties.

What Happens When You Use JDeveloper to Create a Managed Bean

When you create a managed bean and elect to generate the Java file, JDeveloper creates a stub class with the given name and a default constructor. Following is the code that would be added to the `MyBean` class stored in the view package.

```
package view;

public class MyBean {
    public MyBean() {
    }
}
```

You now must add the logic required by your page. You can then refer to that logic using an EL expression that refers to the `managed-bean-name` given to the managed bean. For example, to access the `myInfo` property on the `my_bean` managed bean, the EL expression would be:

```
#{my_bean.myInfo}
```

JDeveloper also adds a `managed-bean` element to the `faces-config.xml` file, as shown below.

```
<managed-bean>
  <managed-bean-name>my_bean</managed-bean-name>
  <managed-bean-class>view.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

What You May Need to Know About Component Bindings and Managed Beans

To avoid issues with managed beans, if your bean needs to use component binding (through the `binding` attribute on the component), you must store the bean in `request` scope. (If your application uses the Fusion technology stack, then you must store it in `backingBean` scope. See *Using a Managed Bean in a Fusion Web Application in Developing Fusion Web Applications with Oracle Application Development Framework*.) However, there may be circumstances where you can't store the bean in `request` or `backingBean` scope. For example, there may be managed beans that are stored in `session` scope so that they can be deployed in a clustered environment, and therefore must implement the `Serializable` interface. When they are serializable, managed beans that change during a request can be distributed to other servers for fail-over. However, ADF Faces components (and JSF components in general) are not serializable. So if a serialized managed bean attempts to access a component using component binding, the bean will fail serialization because the referenced component cannot be serialized. There are also thread safety issues with components bound to serialized managed beans because ADF Faces components are not thread safe.

When you need to store a component reference to a UI component instance in a backing bean that is not using `request` or `backingBean` scope, you should store a reference to the component instance using the Trinidad `ComponentReference` API. The `UIComponentReference.newUIComponentReference()` method creates a serializable reference object that can be used to retrieve a `UIComponent` instance on the current page. The following shows how a managed bean might use the `UIComponentReference` API to get and set values for a search field.

```
...
private ComponentReference<UIInput> searchField;
...
public void setSearchField(UIInput searchField)
{
  if( this.searchField == null)
    this.searchField = ComponentReference.newUIComponentReference(searchField);
}

public UIInput getSearchField()
{
  return searchField ==null ? null : searchField.getComponent();
}
....
```

Keep the following in mind when using the `UIComponentReference` API:

- The API is thread safe as long as it is called on the request thread.
- The ADF Faces component being passed in must have an ID.
- The reference will break if the component is moved between naming containers or if the ID on any of the ancestor naming containers has changed.

For information about the `UIComponentReference` API, see the Trinidad Javadoc.

Viewing ADF Faces Javadoc

The ADF Faces Javadoc provides information required to work programmatically to develop ADF Faces applications using the ADF Faces API.

Often, when you are working with ADF Faces, you will need to view the Javadoc for ADF Faces classes. You can view Javadoc from within JDeveloper.

How to View ADF Faces Source Code and Javadoc

You can view the ADF Faces Javadoc directly from JDeveloper.

To view Javadoc for a class:

1. From the main menu, choose **Navigate > Go to Javadoc**.
2. In the Go to Javadoc dialog, enter the class name you want to view. If you don't know the exact name, you can begin to type the name and JDeveloper will provide a list of classes that match the name. ADF Faces components are in the `oracle.adf.view.rich` package.

Tip:

When in a Java class file, you can go directly to the Javadoc for a class name reference or for a JavaScript function call by placing your cursor on the name or function and pressing `Ctrl+D`.

Part II

Understanding ADF Faces Architecture

Part II documents how to work with the ADF Faces architecture, including the client-side framework, the lifecycle, events, validation and conversion, and partial page rendering.

Specifically, it contains the following chapters:

- [Using ADF Faces Client-Side Architecture](#)
- [Using the JSF Lifecycle with ADF Faces](#)
- [Handling Events](#)
- [Validating and Converting Input](#)
- [Rerendering Partial Page Content](#)

4

Using ADF Faces Client-Side Architecture

This chapter outlines the ADF Faces client-side architecture. This chapter includes the following sections:

- [About Using ADF Faces Architecture](#)
- [Adding JavaScript to a Page](#)
- [Instantiating Client-Side Components](#)
- [Listening for Client Events](#)
- [Accessing Component Properties on the Client](#)
- [Using Bonus Attributes for Client-Side Components](#)
- [Understanding Rendering and Visibility](#)
- [Locating a Client Component on a Page](#)
- [JavaScript Library Partitioning](#)

About Using ADF Faces Architecture

ADF Faces exposes a complete client-side architecture that wraps the browser specific Document Object Model (DOM) API and renders ADF Faces components as client-side JavaScript objects for internal framework use, for use by ADF Faces component developers and for use by ADF Faces application developers.

ADF Faces extends the JavaServer Faces architecture, adding a client-side framework on top of the standard server-centric model. The majority of ADF Faces components are rendered in HTML that is generated on the server-side for a request. In addition, ADF Faces allows component implementations to extend their reach to the client using a client-side component and event model.

The ADF Faces framework already contains much of the functionality for which you would ordinarily need to use JavaScript. In many cases, you can achieve rich component functionality declaratively, without the use of JavaScript. However, there may be times when you do need to add your own JavaScript, for example custom processing in response to a client-side event. In these cases, you can use the client-side framework.

The JavaScript class that you will interact with most is `AdfUIComponent` and its subclasses. An instance of this class is the client-side representation of a server-side component. You can think of a client-side component as a simple property container with support for event handling. Client-side components primarily exist to add behavior to the page by exposing an API contract for both application developers as well as for the framework itself. It is this contract that allows, among other things, toggling the enabled state of a button on the client.

Each client component has a set of properties (key/value pairs) and a list of listeners for each supported event type. All ADF Faces JavaScript classes are prefixed with `Adf`

to avoid naming conflicts with other JavaScript libraries. For example, `RichButton` has `AdfButton`, `RichDocument` has `AdfRichDocument`, and so on.

In the client-side JavaScript layer, client components exist mostly to provide an API contract for the framework and for developers. Because client components exist only to store state and provide an API, they have no direct interaction with the document object model (DOM) whatsoever. All DOM interaction goes through an intermediary called the **peer**. Peers interact with the DOM generated by the Java renderer and handle updating that state and responding to user interactions.

Peers have a number of other responsibilities, including:

- DOM initialization and cleanup
- DOM event handling
- Geometry management
- Partial page response handling
- Child visibility change handling

This separation isolates the component and application developer from changes in the DOM implementation of the component and also isolates the need for the application to know whether a component is implemented in HTML DOM at all (for example the Flash components).

In JSF, as in most component-based frameworks, an intrinsic property of the component model is that components can be nested to form a hierarchy, typically known as the component tree. This simply means that parent components keep track of their children, making it possible to walk over the component tree to find all descendents of any given component. While the full component tree exists on the server, the ADF Faces client-side component tree is sparsely populated.

For performance optimization, client components exist only when they are required, either due to having a `clientListener` handler registered on them, or because the page developer needs to interact with a component on the client side and has specifically configured the client component to be available. You don't need to understand the client framework as except for exceptional cases, you use most of the architectural features declaratively, without having to create any code.

For example, because the framework does not create client components for every server-side component, there may be cases where you need a client version of a component instance. [Instantiating Client-Side Components](#), explains how to do this declaratively. You use the Properties window in JDeveloper to set properties that determine whether a component should be rendered at all, or simply be made not visible, as described in [Understanding Rendering and Visibility](#).

**Note:**

It is also possible for JavaScript components to be present that do not correspond to any existing server-side component. For example, some ADF Faces components have client-side behavior that requires popup content. These components may create `AdfRichPopup` JavaScript components, even though no server-side Java `RichPopup` component may exist.

Other functionality may require you to use the ADF Faces JavaScript API. For example, [Locating a Client Component on a Page](#), explains how to use the API to locate a specific client-side component, and [Accessing Component Properties on the Client](#), documents how to access specific properties.

A common issue with JavaScript-heavy frameworks is determining how best to deliver a large JavaScript code base to the client. If all the code is in a single JavaScript library, there will be a long download time, while splitting the JavaScript into too many libraries will result in a large number of roundtrips. To help mitigate this issue, ADF Faces aggregates its JavaScript code into partitions. A JavaScript library partition contains code for components and/or features that are commonly used together. See [JavaScript Library Partitioning](#).

Adding JavaScript to a Page

ADF Faces exposes JavaScript APIs that application developers use to implement client-side development use cases. To use JavaScript on an ADF Faces view, you can either add JavaScript code to the page source or reference it in an external library file.

You can either add inline JavaScript directly to a page or you can import JavaScript libraries into a page. When you import libraries, you reduce the page content size, the libraries can be shared across pages, and they can be cached by the browser. You should import JavaScript libraries whenever possible. Use inline JavaScript only for cases where a small, page-specific script is needed.

Performance Tip:

Inline JavaScript can increase response payload size, will never be cached in browser, and can block browser rendering. Instead of using inline JavaScript, consider putting all scripts in JavaScript libraries.

If you must use inline JavaScript, include it only in the pages that need it. However, if you find that most of your pages use the same JavaScript code, you may want to consider including the script or import the library in a template.

If a JavaScript code library becomes too big, you should consider splitting it into meaningful pieces and include only the pieces needed by the page (and not placing it in a template). This approach will provide improved performance, because the browser cache will be used and the HTML content of the page will be smaller.

How to Use Inline JavaScript

Create the JavaScript on the page and then use a `clientListener` tag to invoke it.

Before you begin:

It may be helpful to have an understanding of adding JavaScript to a page. For information, see [Adding JavaScript to a Page](#).

To use inline JavaScript:

1. Add the MyFaces Trinidad tag library to the root element of the page by adding the code shown in bold.

```
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
        xmlns:trh="http://myfaces.apache.org/trinidad/html">
```

2. In the Components window, from the Layout panel, in the Core Structure group, drag and drop a **Resource** onto the page.

 **Note:**

Do not use the `f:verbatim` tag in a page or template to specify the JavaScript.

3. In the Insert Resource dialog, select **javascript** from the dropdown menu and click **OK**.

4. Create the JavaScript on the page within the `<af:resource>` tag.

For example, a `sayHello` function might be included in a JSF page like this:

```
<af:resource>
  function sayHello()
  {
    alert("Hello, world!")
  }
</af:resource>
```

5. In the Structure window, right-click the component that will invoke the JavaScript, and choose **Insert inside component > ADF Faces > Client Listener**.
6. In the Insert Client Listener dialog, in the **Method** field, enter the JavaScript function name. In the **Type** field, select the event type that should invoke the function.

How to Import JavaScript Libraries

Use the `af:resource` tag to access a JavaScript library from a page. This tag should appear inside the `document` tag's `metaContainer` facet.

Before you begin:

It may be helpful to have an understanding of adding JavaScript to a page. For information, see [Adding JavaScript to a Page](#).

To access a JavaScript library from a page:

1. Below the `document` tag, add the code shown in bold below and replace `/mySourceDirectory` with the relative path to the directory that holds the JavaScript library.

```
<af:document>
  <f:facet name="metaContainer">
    <af:resource source="/mySourceDirectory"/>
  </facet>
  <af:form></af:form>
</af:document>
```

2. In the Structure window, right-click the component that will invoke the JavaScript, and choose **Insert inside component > ADF Faces > Client Listener**.
3. In the Insert Client Listener dialog, in the **Method** field, enter the name of the function. In the **Type** field, select the event type that should invoke the function.

What You May Need to Know About Accessing Client Event Sources

Often when your JavaScript needs to access a client component, it is within the context of a listener and must access the event's source component. Use the `getSource()` method to get the client component. The following example shows the `sayHello` function accessing the source client component in order to display its name.

```
function sayHello(actionEvent)
{
    var component=actionEvent.getSource();

    //Get the ID for the component
    var id=component.getId();

    alert("Hello from "+id);
}
```

For information about accessing client event sources, see [Using JavaScript for ADF Faces Client Events](#). For information about accessing client-side properties, see [Accessing Component Properties on the Client](#). For a complete description of how client events are handled at runtime, see [What Happens at Runtime: How Client-Side Events Work](#).

Instantiating Client-Side Components

The ADF Faces framework automatically handles JavaScript object instantiation. This supports interacting with the client components and allows developers to manually configure a component to have a client side instance.

By default, the framework does not make any guarantees about which components will have corresponding client-side component instances. To interact with a component on the client, you will usually register a `clientListener` handler. When a component has a registered `clientListener` handler, it will automatically have client-side representation. You can also explicitly configure a component to be available on the client by setting the `clientComponent` attribute to `true`.

How to Configure a Component to for a Client-Side Instance

You can manually configure a component to have a client side instance using the `clientComponent` attribute.

Performance Tip:

Only set `clientComponent` to `true` if you plan on interacting with the component programmatically on the client.

 **Note:**

When the framework creates a client component for its own uses, that client component may only contain information the framework needs at that time. For example, not all of the attributes may be available.

Before you begin:

It may be helpful to have an understanding of client-side instances. See [Instantiating Client-Side Components](#).

To configure a component for a client-side instance:

1. In the Structure window, select the component that needs a client-side instance.
2. In the Properties window, set **ClientSide** to `true`.

What Happens When You Set `clientComponent` to `true`

When you set the `clientComponent` attribute to `true`, the framework creates an instance of an `AdfUIComponent` class for the component. This class provides the API that you can work with on the client side and also provides basic property accessor methods (for example, `getProperty()` and `setProperty()`), event listener registration, and event delivery-related APIs. The framework also provides renderer-specific subclasses (for example, `AdfRichOutputText`) which expose property-specific accessor methods (for example, `getText()` and `setText()`). These accessor methods are simply wrappers around the `AdfUIComponent` class's `getProperty()` and `setProperty()` methods and are provided for coding convenience.

For example, suppose you have an `outputText` component on the page that will get its value (and therefore the text to display) from the `sayHello` function. That function must be able to access the `outputText` component in order to set its value. For this to work, there must be a client-side version of the `outputText` component, as shown in the following JSF page code. Note that the `outputText` component has an `id` value and the `clientComponent` attribute is set to `true`. Also, note there is no value in the example, because that value will be set by the JavaScript.

```
<af:button text="Say Hello">
  <af:clientListener method="sayHello" type="action"/>
</af:button>

<af:outputText id="greeting" value="" clientComponent="true">
```

Because the `outputText` component will now have client-side representation, the JavaScript will be able to locate and work with it.

 **Note:**

The ADF Faces framework may create client components for its own purposes, even when there are no client listeners or when the `clientComponent` attribute is not set to `true`. However, these client components may not be fully functional.

Listening for Client Events

ADF Faces client component objects can be created by adding the `af:clientListener` tag, which is a declarative way to register a client-side listener script to be executed when a specific event type occurs. You can apply the client listener to programmatically created components or to components created for its own use, and also to control navigating away from a JSF page.

In a traditional JSF application, if you want to process events on the client, you must listen to DOM-level events. However, these events are not delivered in a portable manner. The ADF Faces client-side event model is similar to the JSF events model, but implemented on the client. The client-side event model abstracts from the DOM, providing a component-level event model and lifecycle, which executes independently of the server. Consequently, you do not need to listen for `click` events on buttons. You can instead listen for `AdfActionEvent` events, which can be caused by key or mouse events.

Events sent by clients are all subclasses of the `AdfBaseEvent` class. Each client event has a source, which is the component that triggered the event. Events also have a type (for example, `action` or `dialog`), used to determine which listeners are interested in the event. You register a client listener on the component declaratively using the `af:clientListener` tag.

How to Listen for Client Events

You use the `af:clientListener` tag to call corresponding Javascript in response to a client event. For example, suppose you have a button that, in response to a click, should display a "Hello World" alert. You need to first create the JavaScript function that will respond to the event by displaying the alert. You then add the client listener to the component that will invoke that function.

Before you begin:

It may be helpful to have an understanding of client event processing. See [Listening for Client Events](#).

To listen for a client event:

1. Implement the JavaScript function. For example, to display the alert, you might create the JavaScript function shown below.

```
function sayHello(event)
{
    alert("Hello, world!")
}
```

2. In the Components window, from the Operations panel, in the Listeners group, drag and drop a **Client Listener** as a child to the component that will raise the event.

Enter the function created in Step 1, as well as the type of action that the listener should respond to. The following example shows the code that would be created for the listener for the `sayHello` function.

```
<af:button text="Say Hello">
  <af:clientListener method="sayHello" type="action"/>
</af:button>
```

**Tip:**

Because the button has a registered client listener, the framework will automatically create a client version of the component.

When the button is clicked, because there is a client version of the component, the `AdfAction` client event is invoked. Because a `clientListener` tag is configured to listen for the `AdfAction` event, it causes the `sayHello` function to execute. For information about client-side events, see [Using JavaScript for ADF Faces Client Events](#).

How to Use an ADF Client Listener to Control Navigating Away From a JSF Page

When the user attempts to close the browser that displays a JSF page in an ADF Faces application, you can control navigation away from a page by displaying a native browser confirmation dialog to confirm the navigation. To handle this type of event, the `af:document` component supports calling an ADF client listener of type `beforeunload` and ensures that the ADF Faces application will not result in an undefined order of execution that can cause issues in some browsers. Specifically, do not handle this type of navigation event using the browser native `beforeunload` handler like `window.addEventListener("beforeunload", function(e) {...})`.

With support for the `onbeforeunload` event in Oracle ADF, you can define a client event listener to call an event handler on the `af:document` component. Your ADF event handler will be called before leaving or reloading the current page occurs, and the user may cancel the exit entirely within the execution of the ADF Faces framework. If the user continues to navigate away from the page, the browser will raise the `window.unload` event and the ADF Faces application will terminate.

For example, your client listener can handle the ADF `beforeunload` event in a manner similar to how the handler for a browser native `beforeunload` handler operates and should return a string that prompts the user to cancel the exit.

```
...
<af:document>
  <af:resource type="javascript">
    function myBeforeUnload(beforeunloadEvent)
    {
      return "Are you sure that you want to leave?"
    }
  </af:resource>
  <af:clientListener type="beforeunload" method="myBeforeUnload"/>
</af:document>
```

When your ADF `beforeunload` handler is called, if it returns a value other than undefined, then a browser controlled warning dialog will be displayed asking whether to actually navigate away from the current page. Depending on the browser, the returned string may be displayed as part of the warning dialog. The warning message may also be set by calling the `setWarningMessage` method on the ADF `beforeunload` object passed to the handler.

Accessing Component Properties on the Client

ADF Faces extends the JavaServer Faces architecture, adding a client-side framework on top of the standard server-centric model. You can get or set component attribute properties on the client using JavaScript.

For each built-in property on a component, convenience accessor methods are available on the component class. For example, you can call the `getValue()` method on a client component and receive the same value that was used on the server.

 **Note:**

All client properties in ADF Faces use the `getXYZ` function naming convention including boolean properties. The `isXYZ` naming convention for boolean properties is not used.

Constants are also available for the property names on the class object. For instance, you can use `AdfRichDialog.STYLE_CLASS` constant instead of using "styleClass".

 **Note:**

In JavaScript, it is more efficient to refer to a constant than to code the string, as in some JavaScript execution environments, the latter requires an object allocation on each invocation.

When a component's property changes, the end result should be that the component's DOM is updated to reflect its new state, in some cases without a roundtrip to the server. The component's role in this process is fairly limited: it simply stores away the new property value and then notifies the peer of the change. The peer contains the logic for updating the DOM to reflect the new component state.

 **Note:**

Not all property changes are handled through the peer on the client side. Some property changes are propagated back to the server and the component is re-rendered using PPR.

Most property values that are set on the client result in automatic synchronization with the server (although some complex Java objects are not sent to the client at all). There are however, two types of properties that act differently: **secured** properties and **disconnected** properties.

Secured properties are those that cannot be set on the client at all. For example, say a malicious client used JavaScript to set the `immediate` flag on a `Link` component to `true`. That change would then be propagated to the server, resulting in server-side validation being skipped, causing a possible security hole (for information about using

the `immediate` property, see [Using the Immediate Attribute](#)). Consequently, the `immediate` property is a secured property. Attempts to set secured property from JavaScript will fail. See [How to Unsecure the disabled Property](#).

ADF Faces does allow you to configure the `disabled` property so that it can be made unsecure. This can be useful when you need to use JavaScript to enable and disable buttons. See [Secure Client Properties](#) for a list of the ADF Faces secure client properties.

Disconnected properties are those that can be set on the client, but that do not propagate back to the server. These properties have a lifecycle on the client that is independent of the lifecycle on the server. For example, client form input components (like `AdfRichInputText`) have a `submittedValue` property, just as the Java `EditableValueHolder` components do. However, setting this property does not directly affect the server. In this case, standard form submission techniques handle updating the submitted value on the server.

A property can be both disconnected and secured. In practice, such properties act like disconnected properties on the client: they can be set on the client, but will not be sent to the server. But they act like secured properties on the server, in that they will refuse any client attempts to set them.

Secure Client Properties

Secure properties are those that cannot be set on the client, preventing a security hole. [Table 4-1](#) shows the secure properties on the client components.

Table 4-1 Secure Client Properties

Component	Secure Property
<code>AdfRichChooseColor</code>	<code>colorData</code>
<code>AdfRichComboboxListOfValue</code>	<code>disabled</code> <code>readOnly</code>
<code>AdfRichCommandButton</code>	<code>disabled</code> <code>readOnly</code> <code>blocking</code>
<code>AdfRichCommandImageLink</code>	<code>blocking</code> <code>disabled</code> <code>partialSubmit</code>
<code>AdfRichCommandLink</code>	<code>readOnly</code>
<code>AdfRichDialog</code>	<code>dialogListener</code>
<code>AdfRichDocument</code>	<code>failedConnectionText</code>
<code>AdfRichInputColor</code>	<code>disabled</code> <code>readOnly</code> <code>colorData</code>
<code>AdfRichInputDate</code>	<code>disabled</code> <code>readOnly</code> <code>valuePassThru</code>

Table 4-1 (Cont.) Secure Client Properties

Component	Secure Property
AdfRichInputFile	disabled
	readOnly
AdfRichInputListOfValues	disabled
	readOnly
AdfRichInputNumberSlider	disabled
	readOnly
AdfRichInputNumberSpinBox	disabled
	readOnly
	maximum
	minimum
AdfRichInputRangeSlider	stepSize
	disabled
	readOnly
AdfRichInputText	disabled
	readOnly
	secret
AdfRichPopUp	launchPopupListener
	model
	returnPopupListener
	returnPopupDataListener
	createPopupId
AdfRichUIQuery	conjunctionReadOnly
	model
	queryListener
AdfRichSelectBooleanCheckbox	queryOperationListener
	disabled
AdfRichSelectBooleanRadio	readOnly
	disabled
AdfRichSelectManyCheckbox	readOnly
	valuePassThru
	disabled
AdfRichSelectManyChoice	readOnly
	valuePassThru
	disabled
AdfRichSelectManyListBox	readOnly
	valuePassThru
	disabled

Table 4-1 (Cont.) Secure Client Properties

Component	Secure Property
AdfRichSelectManyShuttle	disabled
	readOnly
	valuePassThru
AdfRichSelectOneChoice	disabled
	readOnly
	valuePassThru
AdfRichSelectOneListBox	disabled
	readOnly
	valuePassThru
AdfRichSelectOneRadio	disabled
	readOnly
	valuePassThru
AdfRichSelectOrderShuttle	disabled
	readOnly
	valuePassThru
AdfRichUITable	filterModel
AdfRichTextEditor	disabled
	readOnly
AdfUIChart	chartDrillDownListener
AdfUIColumn	sortProperty
AdfUICommand	actionExpression
	returnListener
	launchListener
	immediate
AdfUIComponentRef	componentType
AdfUIEditableValueBase	immediate
	valid
	required
	localValueSet
	submittedValue
	requiredMessageDetail
AdfUIMessage.js	for
AdfUINavigationLevel	level
AdfUINavigationTree	rowDisclosureListener
	startLevel
	immediate
AdfUIPage	rowDisclosureListener
	immediate

Table 4-1 (Cont.) Secure Client Properties

Component	Secure Property
AdfUIPoll	immediate
	pollListener
AdfUIProgress	immediate
AdfUISelectBoolean	selected
AdfUISelectInput	actionExpression
	returnListener
AdfUISelectRange	immediate
	rangeChangeListener
AdfUIShowDetailBase	immediate
	disclosureListener
AdfUISingleStep	selectedStep
	maxStep
AdfUISubform	default
AdfUITableBase	rowDisclosureListener
	selectionListener
	immediate
	sortListener
	rangeChangeListener
	showAll
AdfUITreeBase	immediate
	rowDisclosureListener
	selectionListener
	focusRowKey
	focusListener
AdfUITreeTable	rangeChangeListener
AdfUIValueBase	converter

How to Set Property Values on the Client

The ADF Faces framework provides `setXYZ` convenience functions that call through to the underlying `ADFUIComponent.setProperty` function, passing the appropriate property name (for more information, see the *JavaScript API Reference for Oracle ADF Faces*). The following code shows how you might use the `setProperty` function to set the `backgroundColor` property on an `inputText` component to red when the value changes.

```
<af:form>
  <af:resource type="javascript">
    function color(event) {
      var inputComponent = event.getSource();
      inputComponent.setProperty("inlineStyle", "background-color:Red");
    }
  </af:resource>
</af:form>
```

```
</af:resource>
<af:outputText id="it" label="label">
  <af:clientListener method="color" type="valueChange"/>
</af:inputText>
</af:form>
```

By using these functions, you can change the value of a property, and as long as it is not a disconnected property or a secure property, the value will also be changed on the server.

What Happens at Runtime: How Client Properties Are Set on the Client

Calling the `setProperty()` function on the client sets the property to the new value, and synchronously fires a `PropertyChangeEvent` event with the old and new values (as long as the value is different). Also, setting a property may cause the component to rerender itself.

How to Unsecure the disabled Property

By default, the disabled property is a secure property. That is, JavaScript cannot be used to set it on the client. However, you can use the `unsecured` property to set the disabled property to be unsecure. You need to manually add this property and the value of `disabled` to the code for the component whose disabled property should be unsecure. For example, the code for a button whose disabled property should be unsecured would be:

```
<af:button text="commandButton 1" id="cb1" unsecure="disabled"/>
```

When the `disabled` attribute was secured, the application could count on the `disabled` attribute to ensure that an action event was never delivered when it shouldn't be. Therefore, when you do unsecure the disabled attribute, the server can no longer count on the disabled attribute being correct and so you must to perform the equivalent check using an `actionListeners`.

For example, say you have an expense approval page, and on that page, you want certain managers to be able to only approve invoices that are under \$200. For this reason, you want the approval button to be disabled unless the current user is allowed to approve the invoice. Without unsecuring the `disabled` attribute, the approval button would remain disabled until a round-trip to the server occurs, where logic determines if the current user can approve the expense. This means that upon rendering, the button may not display correctly for the current user. To have the button to display correctly as the page loads, you need to set the `unsecure` attribute to `disabled` and then use JavaScript on the client to determine if the button should be disabled. But now, *any* JavaScript (including malicious JavaScript that you have no control over) can do the same thing.

To avoid the malicious JavaScript, use the `actionListener` on the button to perform the logic on the server. Adding the logic to the server ensures that the disabled attribute does not get changed when it should not. In the expense approval example, you might have JavaScript that checks that the amount is under \$200, and also use the `actionListener` on the button to recheck that the current manager has the appropriate spending authority before performing the approval.

Similarly, if you allow your application to be modified at runtime, and you allow users to potentially edit the `unsecure` and/or the `disabled` attributes, you must ensure that your application still performs the same logic as if the round-trip to the server had occurred.

How to "Unsynchronize" Client and Server Property Values

There may be cases when you do not want the value of the property to always be delivered and synchronized to the server. For example, say you have `inputText` components in a form, and as soon as a user changes a value in one of the components, you want the changed indicator to display. To do this, you might use JavaScript to set the `changed` attribute to `true` on the client component when the `valueChangeEvent` event is delivered. Say also, you do not want the changed indicator to display once the user submits the page, because at that time, the values are saved.

Say you use JavaScript to set the `changed` attribute to `true` when the `valueChangeEvent` is delivered, like this:

```
<af:form>
  <af:resource type="javascript">
    function changed(event) {
      var inputComponent = event.getSource();
      inputComponent.setChanged(true);
    }
  </af:resource>
  <af:inputText id="it" label="label">
    <af:clientListener method="changed" type="valueChange"/>
  </af:inputText>
  <af:button text="Submit"/>
</af:form>
```

Using this example, the value of the `changed` attribute, which is `true`, will also be sent to the server, because all the properties on the component are normally synchronized to the server. So the changed indicator will continue to display.

To make it so the indicator does not display when the values are saved to the server, you might use one of the following alternatives:

- Move the logic from the client to the server, using an event listener. Use this alternative when there is an event being delivered to the server, such as the `valueChangeEvent` event, as shown here:

```
<af:form>
  <af:inputText label="label"
    autoSubmit="true"
    changed="#{test.changed}"
    valueChangeListener="#{test.valueChange}"/>
  <af:button text="Submit" />
</af:form>
```

This example shows the corresponding managed bean code.

```
import javax.faces.event.ValueChangeEvent;
import oracle.adf.view.rich.context.AdfFacesContext;

public class TestBean {
  public TestBean() {}

  public void valueChange(ValueChangeEvent valueChangeEvent)
  {
```

```

        setChanged(true);
        AdfFacesContext adfFacesContext = AdfFacesContext.getCurrentInstance();
        adfFacesContext.addPartialTarget(valueChangeEvent.getComponent());
        FacesContext.getCurrentInstance().renderResponse();
    }

    public void setChanged(boolean changed)
    {
        _changed = changed;
    }

    public boolean isChanged()
    {
        return _changed;
    }
    private boolean _changed;
}

```

- Move the logic to the server, using JavaScript that invokes a custom server event and a `serverListener` tag, as shown below. Use this when there is no event being delivered.

```

<af:form>
  <af:resource type="javascript">
    function changed(event)
    {
      var inputComponent = event.getSource();
      AdfCustomEvent.queue(inputComponent, "myCustomEvent", null, true);
    }
  </af:resource>
  <af:inputText label="label" changed="#{test2.changed}">
    <af:serverListener type="myCustomEvent"
      method="#{test2.doCustomEvent}"/>
    <af:clientListener method="changed" type="valueChange"/>
  </af:inputText>
  <af:button text="Submit"/>
</af:form>

```

The following example shows the managed bean code.

```

package test;

import javax.faces.context.FacesContext;

import oracle.adf.view.rich.context.AdfFacesContext;
import oracle.adf.view.rich.render.ClientEvent;

public class Test2Bean
{
    public Test2Bean()
    {
    }

    public void doCustomEvent(ClientEvent event)
    {
        setChanged(true);
        AdfFacesContext adfFacesContext = AdfFacesContext.getCurrentInstance();
        adfFacesContext.addPartialTarget(event.getComponent());
        FacesContext.getCurrentInstance().renderResponse();
    }

    public void setChanged(boolean changed)

```

```
{
    _changed = changed;
}

public boolean isChanged()
{
    return _changed;
}
private boolean _changed;
}
```

- On the client component, set the `changed` attribute to `true`, which will propagate to the server, but then use an `actionListener` on the command component to set the `changed` attribute back to `false`. Here is the JSF code:

```
<af:form>
  <af:resource type="javascript">
    function changed(event) {
      var inputComponent = event.getSource();
      inputComponent.setChanged(true);
    }
  </af:resource>
  <af:inputText binding="#{test3.input}" label="label">
    <af:clientListener method="changed" type="valueChange"/>
  </af:inputText>
  <af:button text="Submit" actionListener="#{test3.clear}"/>
</af:form>
```

The following sample shows the corresponding managed bean code.

```
package test;

import javax.faces.event.ActionEvent;

import oracle.adf.view.rich.component.rich.input.RichInputText;

public class Test3Bean
{
    public Test3Bean()
    {
    }

    public void clear(ActionEvent actionEvent)
    {
        _input.setChanged(false);
    }

    public void setInput(RichInputText input)
    {
        _input = input;
    }

    public RichInputText getInput()
    {
        return _input;
    }

    private RichInputText _input;
}
```

Using Bonus Attributes for Client-Side Components

In ADF Faces, bonus attributes are used to return values from server to client. You can create bonus attributes to a component using the Components window in JDeveloper.

In some cases you may want to send additional information to the client beyond the built-in properties. This can be accomplished using bonus attributes. Bonus attributes are extra attributes that you can add to a component using the `clientAttribute` tag. For performance reasons, the only bonus attributes sent to the client are those specified by `clientAttribute`.

The `clientAttribute` tag specifies a name/value pair that is added to the server-side component's attribute map. In addition to populating the server-side attribute map, using the `clientAttribute` tag results in the bonus attribute being sent to the client, where it can be accessed through the `AdfUIComponent.getProperty("bonusAttributeName")` method.

The framework takes care of marshalling the attribute value to the client. The marshalling layer supports marshalling of a range of object types, including strings, booleans, numbers, dates, arrays, maps, and so on. For information on marshalling, see [What You May Need to Know About Marshalling and Unmarshalling Data](#).

Performance Tip:

In order to avoid excessive marshalling overhead, use client-side bonus attributes sparingly.

Note:

The `clientAttribute` tag should be used only for bonus (application-defined) attributes. If you need access to standard component attributes on the client, instead of using the `clientAttribute` tag, simply set the `clientComponent` attribute to `true`. See [Instantiating Client-Side Components](#).

How to Create Bonus Attributes

You can use the Components window to add a bonus attribute to a component.

Before you begin:

It may be helpful to have an understanding of bonus attributes. See [Using Bonus Attributes for Client-Side Components](#).

To create bonus attributes:

1. In the Structure window, select the component to which you would like to add a bonus attribute.

2. In the Components window, from the Operations panel, drag and drop a **Client Attribute** as a child to the component.
3. In the Properties window, set the **Name** and **Value** attributes.

What You May Need to Know About Marshalling Bonus Attributes

Although client-side bonus attributes are automatically delivered from the server to the client, the reverse is not true. That is, changing or setting a bonus attribute on the client will have no effect on the server. Only known (nonbonus) attributes are synchronized from the client to the server. If you want to send application-defined data back to the server, you should create a custom event. See [Sending Custom Events from the Client to the Server](#).

Understanding Rendering and Visibility

You can show (render) and hide an ADF Faces component dynamically on your page, depending on the value of another component. You must set a component's attribute property to true or false to be a part of JSF tree and to generate HTML code for that component.

All ADF Faces display components have two attributes that relate to whether or not the component is displayed on the page for the user to see: `rendered` and `visible`.

The `rendered` attribute has very strict semantics. When `rendered` is set to `false`, there is no way to show a component on the client without a roundtrip to the server. To support dynamically hiding and showing page contents, the framework adds the `visible` attribute. When set to `false`, the component's markup is available on the client but the component is not displayed. Therefore calls to the `setVisible(true)` or `setVisible(false)` method will, respectively, show and hide the component within the browser (as long as `rendered` is set to `true`), whether those calls happen from Java or from JavaScript. However, because `visible` simply shows and hides the content in the DOM, it doesn't always provide the same visual changes as using the `rendered` would.

Performance Tip:

You should set the `visible` attribute to `false` only when you absolutely need to be able to toggle visibility without a roundtrip to the server, for example in JavaScript. Nonvisible components still go through the component lifecycle, including validation.

If you do not need to toggle visibility only on the client, then you should instead set the `rendered` attribute to `false`. Making a component not rendered (instead of not visible) will improve server performance and client response time because the component will not have client-side representation, and will not go through the component lifecycle.

The following example shows two `outputText` components, only one of which is rendered at a time. The first `outputText` component is rendered when no value has been entered into the `inputText` component. The second `outputText` component is rendered when a value is entered.

```

<af:panelGroupLayout layout="horizontal">
  <af:inputText label="Input some text" id="input"
    value="#{myBean.inputValue}"/>
  <af:button text="Enter"/>
</af:panelGroupLayout>
<af:panelGroupLayout layout="horizontal">
  <af:outputLabel value="You entered:"/>
  <af:outputText value="No text entered" id="output1"
    rendered="#{myBean.inputValue==null}"/>
  <af:outputText value="#{myBean.inputValue}"
    rendered="#{myBean.inputValue !=null}"/>
</af:panelGroupLayout>

```

Provided a component is rendered in the client, you can either display or hide the component on the page using the `visible` property.

The next example shows how you might achieve the same functionality as shown above, but in this example, the `visible` attribute is used to determine which component is displayed (the `rendered` attribute is `true` by default, it does not need to be explicitly set).

```

<af:panelGroupLayout layout="horizontal">
  <af:inputText label="Input some text" id="input"
    value="#{myBean.inputValue}"/>
  <af:button text="Enter"/>
</af:panelGroupLayout>
<af:panelGroupLayout layout="horizontal">
  <af:outputLabel value="You entered:"/>
  <af:outputText value="No text entered" id="output1"
    visible="#{myBean.inputValue==null}"/>
  <af:outputText value="#{myBean.inputValue}"
    visible="#{myBean.inputValue !=null}"/>
</af:panelGroupLayout>

```

However, because using the `rendered` attribute instead of the `visible` attribute improves performance on the server side, you may instead decide to have JavaScript handle the visibility, as shown in the following JavaScript code.

```

function showText()
{
  var output1 = AdfUIComponent.findComponent("output1")
  var output2 = AdfUIComponent.findComponent("output2")
  var input = AdfUIComponent.findComponent("input")

  if (input.getValue() == "")
  {
    output1.setVisible(true);
  }
  else
  {
    output2.setVisible(true)
  }
}

```

How to Set Visibility Using JavaScript

You can create a conditional JavaScript function that can toggle the `visible` attribute of components.

Before you begin:

It may be helpful to have an understanding of how components are displayed. See [Understanding Rendering and Visibility](#).

To set visibility:

1. Create the JavaScript that can toggle the visibility. [Example 4-1](#) shows a script that turns visibility on for one `outputText` component if there is no value; otherwise, the script turns visibility on for the other `outputText` component.
2. For each component that will be needed in the JavaScript function, expand the **Advanced** section of the Properties window and set **ClientComponent** attribute to `true`. This creates a client component that will be used by the JavaScript.
3. For the components whose visibility will be toggled, set the `visible` attribute to `false`.

This example shows the full page code used to toggle visibility with JavaScript.

Example 4-1 JavaScript for Setting Visibility

```
<f:view>
<af:resource>
  function showText()
  {
    var output1 = AdfUIComponent.findComponent("output1")
    var output2 = AdfUIComponent.findComponent("output2")
    var input = AdfUIComponent.findComponent("input")

    if (input.value == "")
    {
      output1.setVisible(true);
    }
    else
    {
      output2.setVisible(true)
    }
  }
</af:resource>
<af:document>
  <af:form>
    <af:panelGroupLayout layout="horizontal">
      <af:inputText label="Input some text" id="input"
        value="#{myBean.inputValue}" clientComponent="true"
        immediate="true"/>
      <af:button text="Enter" clientComponent="true">
        <af:clientListener method="showText" type="action"/>
      </af:button>
    </af:panelGroupLayout>
    <af:panelGroupLayout layout="horizontal">
      <af:outputLabel value="You entered:" clientComponent="false"/>
      <af:outputText value="No text entered" id="output1"
        visible="false" clientComponent="true"/>
      <af:outputText value="#{myBean.inputValue}" id="output2"
        visible="false" clientComponent="true"/>
    </af:panelGroupLayout>
  </af:form>
</af:document>
</f:view>
```

```
</af:form>
</af:document>
</f:view>
```

What You May Need to Know About Visible and the isVisible Function

If the parent of a component has its `visible` attribute set to `false`, when the `isVisible` function is run against a child component whose `visible` attribute is set to `true`, it will return `true`, even though that child is not displayed. For example, say you have a `panelGroupLayout` component that contains an `outputText` component as a child, and the `panelGroupLayout` component's `visible` attribute is set to `false`, while the `outputText` component's `visible` attribute is left as the default (`true`). On the client, neither the `panelGroupLayout` nor the `outputText` component will be displayed, but if the `isVisible` function is run against the `outputText` component, it will return `true`.

For this reason, the framework provides the `isShowing()` function. This function will return `false` if the component's `visible` attribute is set to `false`, or if any parent of that component has `visible` set to `false`.

Locating a Client Component on a Page

When you need to find a client component that is not the source of an event, you can use the `AdfUIComponent.findComponent(expr)` method. This method is similar to the JSF `UIComponent.findComponent()` method, which searches for and returns the `UIComponent` object with an ID that matches the specified search expression.

The `AdfUIComponent.findComponent(expr)` method simply works on the client instead of the server.

The following example shows the `sayHello` function finding the `outputText` component using the component's ID.

```
function sayHello(actionEvent)
{
    var buttonComponent=actionEvent.getSource();

    //Find the client component for the "greeting" af:outputText
    var greetingComponent=buttonComponent.findComponent("greeting");

    //Set the value for the outputText component
    greetingComponent.setValue("Hello World")
}
```

ADF Faces also has the `AdfPage.PAGE.findComponentByAbsoluteId(absolute expr)` method. Use this method when you want to hard-code the String for the ID. Use `AdfUIComponent.findComponent(expr)` when the client ID is being retrieved from the component.

 **Note:**

There is also a confusingly named `AdfPage.PAGE.findComponent(clientId)` method, however this function uses implementation-specific identifiers that can change between releases and should not be used by page authors.

What You May Need to Know About Finding Components in Naming Containers

If the component you need to find is within a component that is a naming container (such as `pageTemplate`, `subform`, `table`, and `tree`), then instead of using the `AdfPage.PAGE.findComponentByAbsoluteId(absolute expr)` method, use the `AdfUIComponent.findComponent(expr)` method. The expression can be either absolute or relative.

 **Tip:**

You can determine whether or not a component is a naming container by reviewing the component tag documentation. The tag documentation states whether a component is a naming container.

Absolute expressions are built as follows:

```
":" + (namingContainersToJumpUp * ":") + some ending portion of the  
clientIdOfComponentToFind
```

For example, to find a table whose ID is `t1` that is within a panel collection component whose ID is `pc1` contained in a region whose ID is `r1` on page that uses the `myTemplate` template, you might use the following:

```
:myTemplate:r1:pc1:t1
```

Alternatively, if both the components (the one doing the search and the one being searched for) share the same `NamingContainer` component somewhere in the hierarchy, you can use a relative path to perform a search relative to the component doing the search. A relative path has multiple leading `NamingContainer.SEPARATOR_CHAR` characters, for example:

```
":" + clientIdOfComponentToFind
```

In the preceding example, if the component doing the searching is also in the same region as the table, you might use the following:

```
::somePanelCollection:someTable
```

 **Tip:**

Think of a naming container as a folder and the `clientId` as a file path. In terms of folders and files, you use two sequential periods and a slash (`../`) to move up in the hierarchy to another folder. This is the same thing that the multiple colon (`:`) characters do in the `findComponent()` expression. A single leading colon (`:`) means that the file path is absolute from the root of the file structure. If there are multiple leading colon (`:`) characters at the beginning of the expression, then the first one is ignored and the others are counted, one set of periods and a slash (`../`) per colon (`:`) character.

Note that if you were to use the `AdfPage.findComponentByAbsoluteId()` method, no leading colon is needed as, the path always absolute.

When deciding whether to use an absolute or relative path, keep the following in mind:

- If you know that the component you are trying to find will always be in the same naming container, then use an absolute path.
- If you know that the component performing the search and the component you are trying to find will always be in the same relative location, then use a relative path.

There are no `getChildren()` or `getFacet()` functions on the client. Instead, the `AdfUIComponent.visitChildren()` function is provided to visit all children components or facets (that is all descendents). Because ADF Faces uses a sparse component tree (that is, client components are created on an as-needed basis, the component that the `getParent()` method might return on the client may not be the actual parent on the server (it could be any ancestor). Likewise, the components that appear to be immediate children on the client could be any descendants. See the *Java API Reference for Oracle ADF Faces*.

JavaScript Library Partitioning

In ADF Faces, JavaScript Partitioning feature helps combine individual JavaScript files from ADF components together into larger collections, thus reducing the number of downloads. You can use the default partition provided by ADF Faces or create/configure a JavaScript partition as required by the application.

A common issue with JavaScript-heavy frameworks is determining how best to deliver a large JavaScript code base to the client. On one extreme, bundling all code into a single JavaScript library can result in a long download time. On the other extreme, breaking up JavaScript code into many small JavaScript libraries can result in a large number of roundtrips. Both approaches can result in the end user waiting unnecessarily long for the initial page to load.

To help mitigate this issue, ADF Faces aggregates its JavaScript code into partitions. A JavaScript library partition contains code for components and/or features that are commonly used together. By default, ADF Faces provides a partitioning that is intended to provide a balance between total download size and total number of roundtrips.

One benefit of ADF Faces's library partitioning strategy is that it is configurable. Because different applications make use of different components and features, the default partitioning provided by ADF Faces may not be ideal for all applications. As

such, ADF Faces allows the JavaScript library partitioning to be customized on a per-application basis. This partitioning allows application developers to tune the JavaScript library footprint to meet the needs of their application.

ADF Faces groups its components' JavaScript files into JavaScript features. A JavaScript feature is a collection of JavaScript files associated with a logical identifier that describes the feature. For example, the `panelStretchLayout` client component is comprised of the following two JavaScript files

- `oracle/adf/view/js/component/rich/layout/ AdfRichPanelStretchLayout.js`
- `oracle/adfinternal/view/js/laf/dhtml/rich/ AdfDhtmlPanelStretchLayoutPeer.js`

These two files are grouped into the `AdfRichPanelStretchLayout` feature.

JavaScript features are further grouped into JavaScript partitions. JavaScript partitions allow you to group JavaScript features into larger collections with the goal of influencing the download size and number of round trips. For example, since the `panelStretchLayout` component is often used with the `panelSplitter` component, the features for these two components are grouped together in the `stretch` partition, along with the other ADF Faces layout components that can stretch their children. At runtime, when a page is loaded, the framework determines the components used on the page, and then from that, determines which features are needed (feature names are the same as the components' constructor name). Only the partitions that contain those features are downloaded.

Features and partitions are defined using configuration files. ADF Faces ships with a default features and partitions configuration file. You can overwrite the default partitions file by creating your own implementation.

By default, JavaScript partitioning is turned on. Whether or not your application uses JavaScript partitioning is determined by a context parameter in the `web.xml` file. See [JavaScript Partitioning](#).

How to Create JavaScript Partitions

You create a JavaScript partition by creating an `adf-js-partitions.xml` file, and then adding entries for the features.

Note:

ADF Faces provides a default `adf-js-partitions.xml` file (see [The adf-js-partitions.xml File](#)). If you want to change the partition configuration, you need to create your own complete `adf-js-partitions.xml` file. At runtime, the framework will search the `WEB-INF` directory for that file. If one is not found, it will load the default partition file.

Before you begin:

It may be helpful to have an understanding of JavaScript partitioning works. For information, see [JavaScript Library Partitioning](#).

To create JavaScript partitions:

1. In the Applications window, right-click **WEB-INF** and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **XML** and then **XML Document**, and click **OK**.

 **Tip:**

If you don't see the **General** node, click the **All Technologies** tab at the top of the Gallery.

3. In the Create XML File dialog, enter `adf-js-partitions.xml` as the file name and save it in the **WEB-INF** directory.
4. In the source editor, replace the generated code with the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<partitions xmlns="http://xmlns.oracle.com/adf/faces/partition">

</partitions>
```

5. Add the following elements to populate a partition with the relevant features.
 - `partitions`: The root element of the configuration file.
 - `partition`: Create as a child to the `partitions` element. This element must contain one `partition-name` child element and one or more `feature` elements.
 - `partition-name`: Create as a child to the `partition` element. Specifies the name of the partition. This value will be used to produce a unique URL for this partition's JavaScript library.
 - `feature`: Create as a child to the `partition` element. Specifies the feature to be included in this partition. There can be multiple `feature` elements.

 **Tip:**

Any feature configured in the `adf-js-features.xml` file that does not appear in a partition is treated as if it were in its own partition.

The following example shows the `partition` element for the `tree` partition that contains the `AdfRichTree` and `AdfRichTreeTable` features.

```
<partition>
  <partition-name>tree</partition-name>
  <feature>AdfUITree</feature>
  <feature>AdfUITreeTable</feature>
  <feature>AdfRichTree</feature>
  <feature>AdfRichTreeTable</feature>
</partition>
```

What Happens at Runtime: JavaScript Partitioning

ADF Faces loads the library partitioning configuration files at application initialization time. First, ADF Faces searches for all `adf-js-features.xml` files in the **META-INF**

directory and loads all that are found (including the ADF Faces default feature configuration file).

For the partition configuration file, ADF Faces looks for a single file named `adf-js-partitions.xml` in the `WEB-INF` directory. If no such file is found, the ADF Faces default partition configuration is used.

During the render traversal, ADF Faces collects information about which JavaScript features are required by the page. At the end of the traversal, the complete set of JavaScript features required by the (rendered) page contents is known. Once the set of required JavaScript features is known, ADF Faces uses the partition configuration file to map this set of features to the set of required partitions. Given the set of required partitions, the HTML `<script>` references to these partitions are rendered just before the end of the HTML document.

5

Using the JSF Lifecycle with ADF Faces

This chapter describes the JSF page request lifecycle and the additions to the lifecycle from ADF Faces, and how to use the lifecycle properly in your application. This chapter includes the following sections:

- [About Using the JSF Lifecycle and ADF Faces](#)
- [Using the Immediate Attribute](#)
- [Using the Optimized Lifecycle](#)
- [Using the Client-Side Lifecycle](#)
- [Using Subforms to Create Sections on a Page](#)
- [Object Scope Lifecycles](#)
- [Passing Values Between Pages](#)

About Using the JSF Lifecycle and ADF Faces

The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page, translated to HTML. You can learn about the different phases of JSF lifecycle and ADF Faces lifecycle on a page request and how the events are processed before and after each phase.

ADF Faces applications use both the JSF lifecycle and the ADF Faces lifecycle. The ADF Faces lifecycle extends the JSF lifecycle, providing additional functionality, such as a client-side value lifecycle, a subform component that allows you to create independent submittable sections on a page without the drawbacks (for example, lost user edits) of using multiple forms on a single page, and additional scopes.

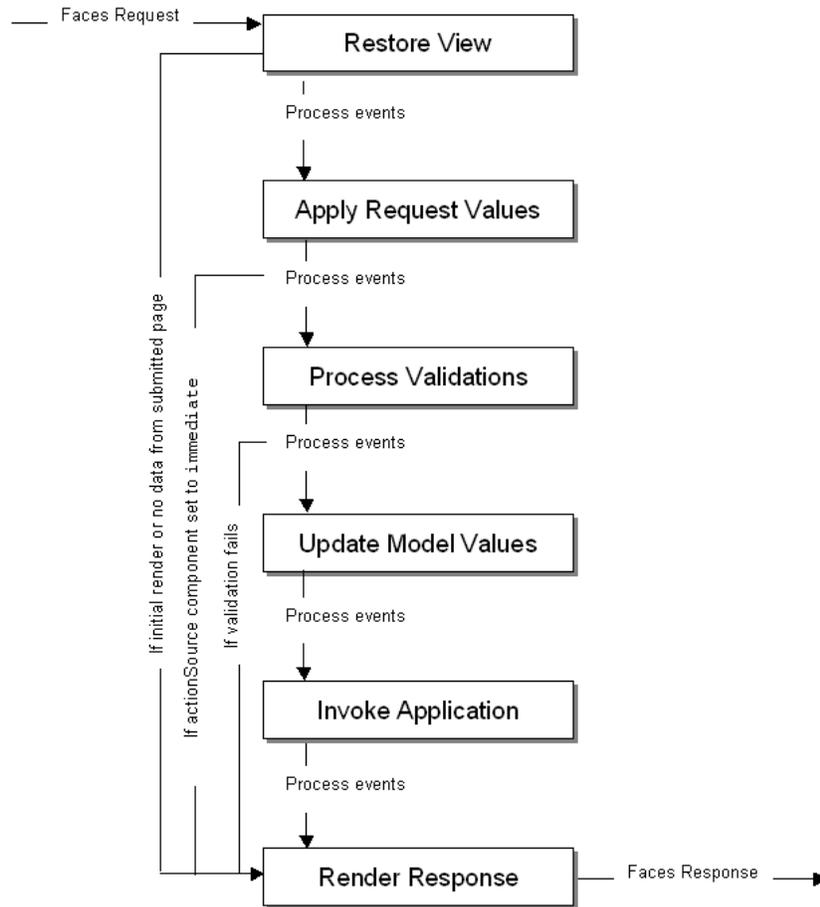
To better understand the lifecycle enhancements that the framework delivers, it is important that you understand the standard JSF lifecycle. This section provides only an overview. For a more detailed explanation, refer to the JSF specification at <http://www.jcp.org/en/jsr/detail?id=314>.

When a JSF page is submitted and a new page is requested, the JSF page request lifecycle is invoked. This lifecycle handles the submission of values on the page, validation for components on the current page, navigation to and display of the components on the resulting page, as well as saving and restoring state. The `FacesServlet` object manages the page request lifecycle in JSF applications. The `FacesServlet` object creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the lifecycle.

The JSF lifecycle phases use a UI component tree to manage the display of the faces components. This tree is a runtime representation of a JSF page: each UI component tag in a page corresponds to a UI component instance in the tree.

[Figure 5-1](#) shows the JSF lifecycle of a page request. As shown, events are processed before and after each phase.

Figure 5-1 Lifecycle of a Page Request in an ADF Faces Application



In a JSF application, the page request lifecycle is as follows:

- **Restore View:** The component tree is established. If this is not the initial rendering (that is, if the page was submitted back to server), the tree is restored with the appropriate state. If this is the initial rendering, the component tree is created and the lifecycle jumps to the Render Response phase.
- **Apply Request Values:** Each component in the tree extracts new values from the request parameters (using its decode method) and stores the values locally. Most associated events are queued for later processing. If a component has its `immediate` attribute set to `true`, then the validation, the conversion, and the events associated with the component are processed during this phase. See [Using the Immediate Attribute](#).
- **Process Validations:** Local values of components are converted from the input type to the underlying data type. If the converter fails, this phase continues to completion (all remaining converters, validators, and required checks are run), but at completion, the lifecycle jumps to the Render Response phase.

If there are no failures, the `required` attribute on the component is checked. If the value is `true`, and the associated field contains a value, then any associated validators are run. If the value is `true` and there is no field value, this phase

completes (all remaining validators are executed), but the lifecycle jumps to the Render Response phase. If the value is `false`, the phase completes, unless no value is entered, in which case no validation is run. For information about conversion and validation, see [Validating and Converting Input](#).

At the end of this phase, converted versions of the local values are set, any validation or conversion error messages and events are queued on the `FacesContext` object, and any value change events are delivered.

 **Tip:**

In short, for an input component that can be edited, the steps for the Process Validations phase is as follows:

1. If a converter fails, the required check and validators are not run.
2. If the converter succeeds but the required check fails, the validators are not run.
3. If the converter and required check succeed, all validators are run. Even if one validator fails, the rest of the validators are run. This is because when the user fixes the error, you want to give them as much feedback as possible about what is wrong with the data entered.

For example suppose you have a `dateTimeRange` validator that accepted dates only in the year 2015 and you had a `dateRestrictionValidator` validator that did not allow the user to pick Sundays. If the user entered November 16, 2014 (a Sunday), you want to give the feedback that this fails both validators to maximize the chance the user will enter valid data.

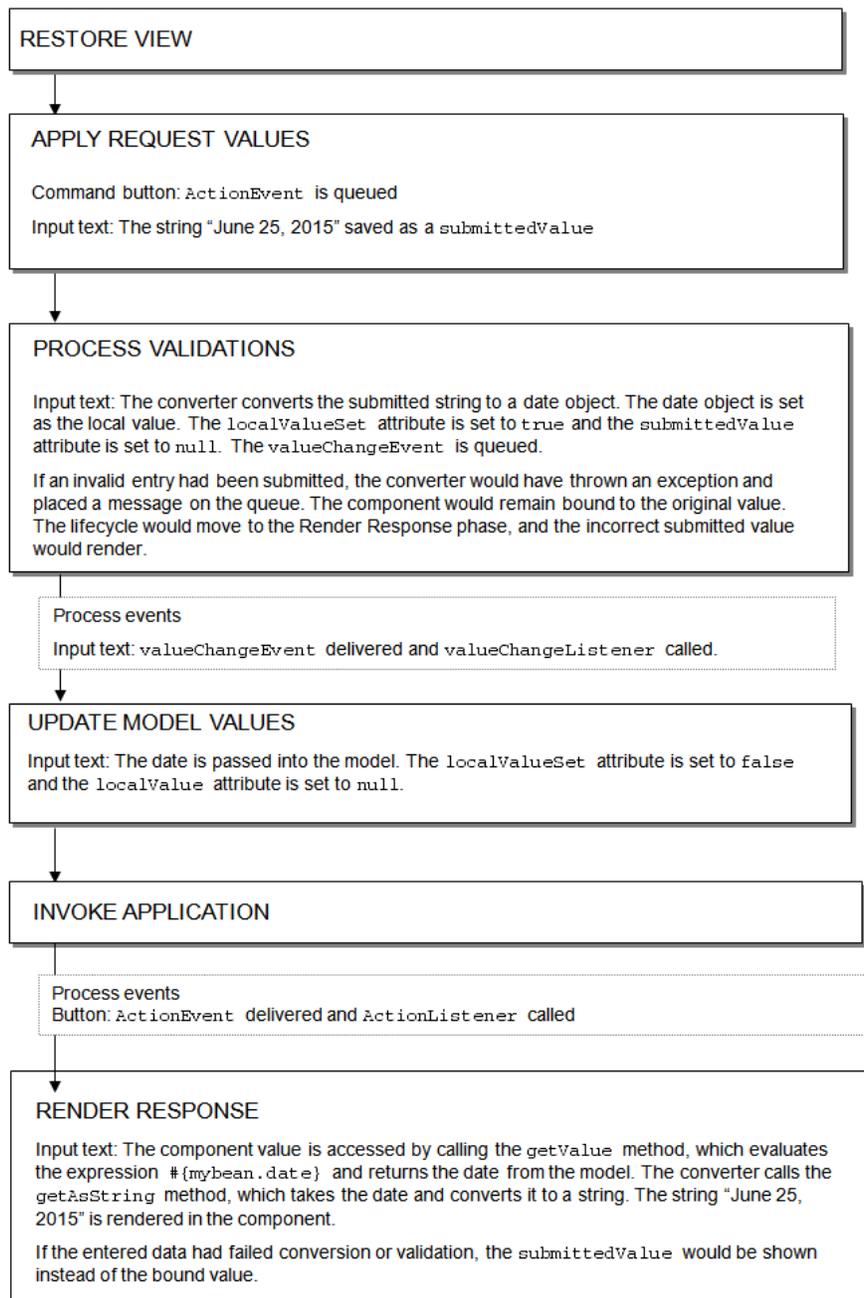
- Update Model Values: The component's validated local values are moved to the model, and the local copies are discarded.
- Invoke Application: Application-level logic (such as event handlers) is executed.
- Render Response: The components in the tree are rendered. State information is saved for subsequent requests and for the Restore View phase.

To help illustrate the lifecycle, consider a page that has a simple input text component where a user can enter a date and then click a button to submit the entered value. A `valueChangeListener` method is also registered on the component. The following shows the code for the example.

```
<af:form>
  <af:inputText value="#{mybean.date}"
    valueChangeListener="#{mybean.valueChangeListener}">
    <af:convertDateTime dateStyle="long"/>
  </af:inputText>
  <af:button text="Save" actionListener="#{mybean.actionListener}"/>
</af:form>
```

Suppose a user enters the string "June 25, 2015" and clicks the submit button. [Figure 5-2](#) shows how the values pass through the lifecycle and where the different events are processed.

Figure 5-2 Example of Values and Events in the JSF Lifecycle



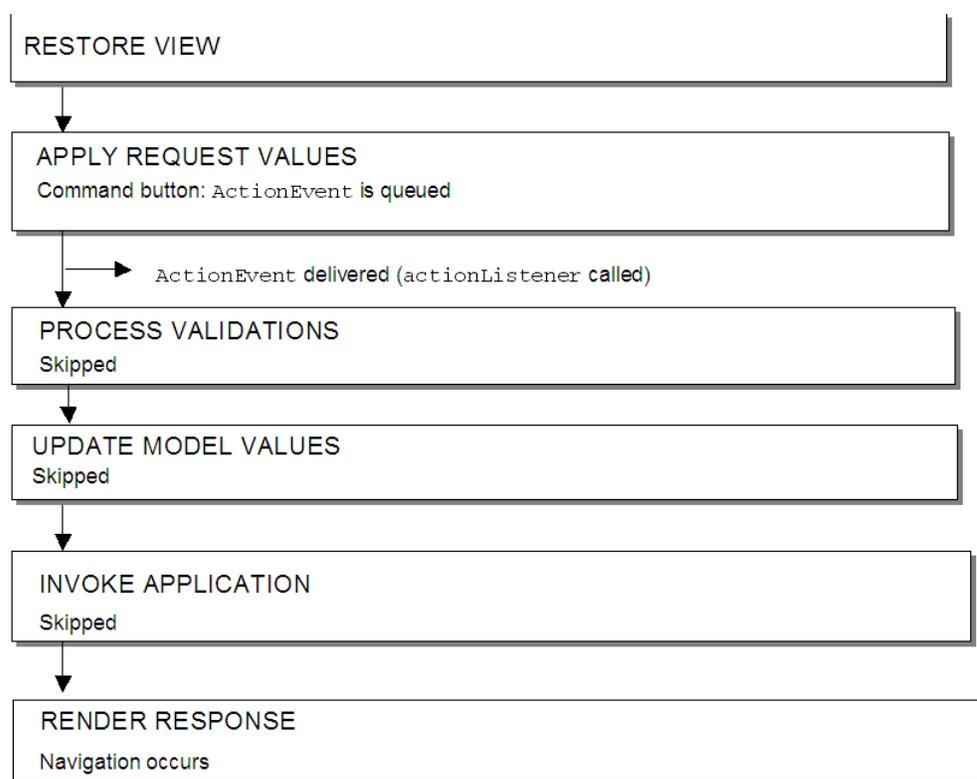
Using the Immediate Attribute

In ADF Faces lifecycle or in JSF lifecycle, when a component's immediate attribute is set to true, the validation, conversion, and events associated with these components are processed during the Apply Request Values phase rather than in a later phase. Use this attribute to navigate to another page without processing any data currently in the input fields of the current screen.

You can use the `immediate` attribute to allow processing of components to move up to the Apply Request Values phase of the lifecycle. When `actionSource` components (such as a `commandButton`) are set to `immediate`, events are delivered in the Apply Request Values phase instead of in the Invoke Application phase. The `actionListener` handler then calls the Render Response phase, and the validation and model update phases are skipped.

For example, you might want to configure a Cancel button to be `immediate`, and have the action return a string used to navigate back to the previous page (for information about navigation, see [Working with Navigation Components](#)). Because the Cancel button is set to `immediate`, when the user clicks the Cancel button, all validation is skipped, any entered data is not updated to the model, and the user navigates as expected, as shown in [Figure 5-3](#).

Figure 5-3 Lifecycle for Button Set to Immediate



 **Note:**

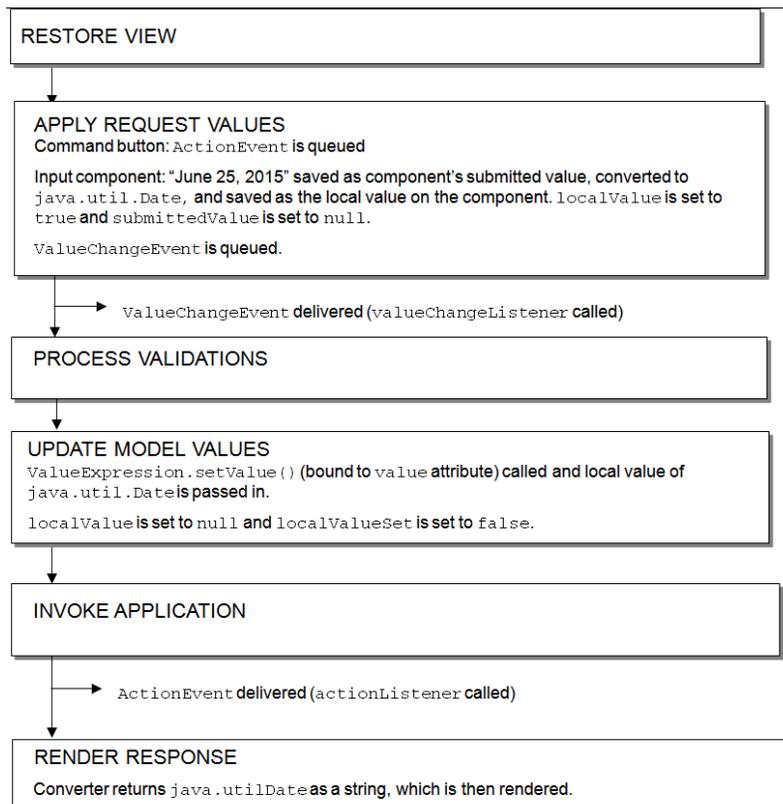
A button that does not provide any navigation and is set to `immediate` will also go directly to the Render Response phase: the Validation, Update Model, and Invoke Application phases are skipped, so any new values will not be pushed to the server.

As with command components, for components that invoke disclosure events, (such as a `showDetail` component), and for `editableValueHolder` components

(components that hold values that can change, such as an `inputText` component), when set to `immediate`, the events are delivered to the Apply Request Values phase. However, for `editableValueHolder` components, instead of skipping phases, conversion, validation, and delivery of `valueChangeEvent`s events for the immediate components are done earlier in the lifecycle, during the Apply Request Values phase, instead of after the Process Validations phase. No lifecycle phases are skipped.

Figure 5-4 shows the lifecycle for an input component whose immediate attribute is set to true. The input component takes a date entered as a string and stores it as a date object when the command button is clicked.

Figure 5-4 Immediate Attribute on an Input Component



Setting `immediate` to `true` for an input component can be useful when one or more input components must be validated before other components. Then, if one of those components is found to have invalid data, validation is skipped for the other input components in the same page, thereby reducing the number of error messages shown for the page.

For example, suppose you have a form used to collect name and address information and you have the **Name** field set to required. The address fields need to change, based on the country chosen in the Country dropdown field. You would set that dropdown field (displayed using a `selectOneChoice` component) to `immediate` so that when the user selects a country, a validation error doesn't occur because the name field is blank. The following example shows the code for the address field (the switching functionality between countries is not shown)

```

<af:form id="f1">
  <af:panelFormLayout id="pf11">
    <af:panelGroupLayout id="pgl2" layout="vertical">
      <af:inputText label="Name" id="it1" value="#{bean.name}"
        required="true"/>
      <af:inputText label="Street" id="it2" value="#{bean.street}"/>
    <af:panelGroupLayout id="pgl1" layout="horizontal">
      <af:inputText label="City" id="it3" value="#{bean.city}"/>
      <af:inputText label="State" id="it4" value="#{bean.state}"/>
      <af:inputText label="Zip" id="it5" value="#{bean.zip}"/>
    </af:panelGroupLayout>
    <af:selectOneChoice label="Country" id="soc1" immediate="true">
      valueChangeListener="bean.countryChanged">
      <af:selectItem label="US" value="US" id="si1"/>
      <af:selectItem label="Canada" value="Canada" id="si2"/>
    </af:selectOneChoice>
  </af:panelFormLayout>
</af:form>

```

In this example, the `selectOneChoice` value change event is fired after the Apply Request Values phase, and so the `countryChanged` listener method is run before the Name `inputText` component is validated. However, remember that for editable components, phases are not skipped - validation will still be run during this request. To prevent validation of the other components, you need to call the `renderResponse` method directly from your listener method. This call skips the rest of the lifecycle, including validation and jumps directly to the Render Response phase.

```

public void countryChanged(ValueChangeEvent event)
  FacesContext context = FacesContext.getCurrentInstance();
  . . .
  some code to change the locale of the form
  . . .
context.renderResponse();

```

Note:

Be careful when calling `RenderResponse()`. Any component that goes through a partial lifecycle will have an incomplete state that may cause trouble. The values on those components are decoded but not validated and pushed to model, so you need to clear their submitted value when calling `RenderResponse()`.

 **Note:**

Only use the `immediate` attribute when the desired functionality is to leave the page and not ever submit any values other than the immediate components. In the Cancel example, no values will ever be submitted on the page. In the address example, we want the page to be rerendered without ever submitting any other values.

The `immediate` attribute causes the components to be decoded (and thus have a submitted value), but not processed further. In the case of an immediate action component, the other components never reach the validation or update model phases, where the submitted value would be converted to a local value and then the local value pushed into the model and cleared. So values will not be available.

Don't use the `immediate` attribute when you want a number of fields to be processed before other fields. When that is the case, it's best to divide your page up into separate sub-forms, so that only the fields in the sub-forms are processed when the lifecycle runs.

There are some cases where setting the `immediate` attribute to true can lead to better performance: When you create a navigation train, and have a `commandNavItem` component in a `navigationPane` component, you should set the `immediate` attribute to true to avoid processing the data from the current page (train stop) while navigating to the next page. For more information, see [How to Create the Train Model](#). If an input component value has to be validated before any other values, the `immediate` attribute should be set to `true`. Any errors will be detected earlier in the lifecycle and additional processing will be avoided.

How to Use the Immediate Attribute

Before you begin:

It may be helpful to have an understanding of the `immediate` attribute. See [Using the Immediate Attribute](#).

To use the `immediate` attribute:

1. On the JSF page, select the component that you want to be immediate.
2. In the Properties window, expand the **Behavior** section and set the `immediate` attribute to `true`.

Using the Optimized Lifecycle

In ADF Faces, Partial Page Rendering (PPR) optimizes the JSF lifecycle. Use the event root components to determine boundaries on the page and to allow the lifecycle to run just on components within that boundary.

ADF Faces provides an optimized lifecycle that runs the JSF page request lifecycle (including conversion and validation) only for certain components within a boundary on a page. This partial page lifecycle is called **partial page rendering** (PPR). Certain ADF Faces components are considered event root components, and are what determine the boundaries on which the optimized lifecycle is run. There are two ways

event root components are determined. First, certain components are always considered event roots. For example, the `panelCollection` component is an event root component. It surrounds a table and provides among other things, a toolbar. Action events triggered by a button on the toolbar will cause the lifecycle to be run only on the child components of the `panelCollection`.

The second way event root components are determined is that certain events designate an event root. For example, the disclosure event sent when expanding or collapsing a `showDetail` component (see [Displaying and Hiding Contents Dynamically](#)) indicates that the `showDetail` component is a root, and so the lifecycle is run only on the `showDetail` component and any child components. For information about PPR and event roots, including a list of event root components, see [Events and Partial Page Rendering](#).

Aside from running on the event root component and its child components, you can declaratively configure other components outside the event root hierarchy to participate in the optimized lifecycle. You can also specifically configure only certain events for a component to trigger the optimized lifecycle, and configure which components will actually execute or will only be refreshed.

For information about how the ADF Faces framework uses PPR, and how you can use PPR throughout your application, see [Rerendering Partial Page Content](#).

Using the Client-Side Lifecycle

The ADF Faces framework provides client-side conversion and validation. You can create your own JavaScript-based converters and validators that run on the page without a trip to the server.

You can use client-side validation so that when a specific client event is queued, it triggers client validation of the appropriate form or subform (for information about subforms, see [Using Subforms to Create Sections on a Page](#)). If this client validation fails, meaning there are known errors, then the events that typically propagate to the server (for example, a button's `actionEvent` when a form is submitted) do not go to the server. Having the event not delivered also means that nothing is submitted and therefore, none of the client listeners are called. This is similar to server-side validation in that when validation fails on the server, the lifecycle jumps to the Render Response phase; the action event, though queued, will never be delivered; and the `actionListener` handler method will never be called.

For example, ADF Faces provides the `required` attribute for input components, and this validation runs on the client. When you set this attribute to `true`, the framework will show an error on the page if the value of the component is `null`, without requiring a trip to the server. The following example shows code that has an `inputText` component's `required` attribute set to `true`, and a button whose `actionListener` attribute is bound to a method on a managed bean.

```
<af:form>
  <af:inputText id="input1" required="true" value="a"/>
  <af:button text="Search" actionListener="#{demoForm.search}"/>
</af:form>
```

When this page is run, if you clear the field of the value of the `inputText` component and tab out of the field, the field will redisplay with a red outline. If you then click into the field, an error message will state that a value is required, as shown in [Figure 5-5](#).

There will be no trip to the server; this error detection and message generation is all done on the client.

Figure 5-5 Client-Side Validation Displays an Error Without a Trip to the Server



In this same example, if you were to clear the field of the value and click the **Search** button, the page would not be submitted because the required field is empty and therefore an error occurs; the action event would not be delivered, and the method bound to the action listener would not be executed. This process is what you want, because there is no reason to submit the page if the client can tell that validation will fail on the server.

For information about using client-side validation and conversion, see [Validating and Converting Input](#).

Using Subforms to Create Sections on a Page

ADF Faces provides the subform component, which adds flexibility by defining subregions whose component values can be submitted separately within a form. The content of a subform can be validated based on certain criteria.

In the JSF reference implementation, if you want to independently submit a section of the page, you have to use multiple forms. However multiple forms require multiple copies of page state, which can result in the loss of user edits in forms that aren't submitted.

ADF Faces adds support for a subform component, which represents an independently submittable section of a page. The contents of a subform will be validated (or otherwise processed) only if a component inside of the subform is responsible for submitting the page, allowing for comparatively fine-grained control of the set of components that will be validated and pushed into the model without the compromises of using entirely separate form elements. When a page using subforms is submitted, the page state is written only once, and all user edits are preserved.

Best Practice:

Always use only a single `form` tag per page. Use the `subform` tag where you might otherwise be tempted to use multiple `form` tags.

A subform will always allow the Apply Request Values phase to execute for its child components, even when the page was submitted by a component outside of the subform. However, the Process Validations and Update Model Values phases will be skipped (this differs from an ordinary form component, which, when not submitted, cannot run the Apply Request Values phase). To allow components in subforms to be processed through the Process Validations and Update Model Value phases when a component outside the subform causes a submit action, use the `default` attribute.

When a subform's `default` attribute is set to `true`, it acts like any other subform in most respects, but if no subform on the page has an appropriate event come from its child components, then any subform with `default` set to `true` will behave as if one of its child components caused the submit. For information about subforms, see [Defining Forms](#).

Object Scope Lifecycles

When an object is created in a JSF application, it defines or defaults to a given scope; this object scope describes how widely it is available and who has access to it. You can use the standard JSF scopes or the ADF Faces scopes that are available for an object in a JSF application.

At runtime, you pass data to pages by storing the needed data in an object scope where the page can access it. The scope determines the lifespan of an object. Once you place an object in a scope, it can be accessed from the scope using an EL expression. For example, you might create a managed bean named `foo`, and define the bean to live in the Request scope. To access that bean, you would use the expression `#{requestScope.foo}`.

There are five types of scopes in a standard JSF application:

- `applicationScope`: The object is available for the duration of the application.
- `sessionScope`: The object is available for the duration of the session.
- `viewScope`: The object is available until the user finishes interaction with the current view. The object is stored in a map on the `UIViewRoot` object. Note that this object is emptied upon page refresh or a redirect to the view.

Tip:

If you need the object to survive a page refresh or redirect to the same view, then use the ADF Faces version of `viewScope`.

- `flashScope`: The object is available during a single view transition, and is cleaned up before moving on to the next view. You can place a parameter value in `flashScope` and it will be available to the resulting page, surviving redirects.
- `requestScope`: The object is available for the duration between the time an HTTP request is sent until a response is sent back to the client.

In addition to the standard JSF scopes, ADF Faces provides the following scopes:

- `pageFlowScope`: The object is available as long as the user continues navigating from one page to another. If the user opens a new browser window and begins navigating, that series of windows will have its own `pageFlowScope` scope.
- `backingBeanScope`: Used for managed beans for page fragments and declarative components only. The object is available for the duration between the time an HTTP request is sent until a response is sent back to the client. This scope is needed because there may be more than one page fragment or declarative component on a page, and to avoid collisions between values, any values must be kept in separate scope instances. Use `backingBeanScope` scope for any managed bean created for a page fragment or declarative component.

- `viewScope`: The object is available until the ID for the current view changes. Use `viewScope` scope to hold values for a given page. Unlike the JSF `viewScope`, objects stored in the ADF Faces `viewScope` will survive page refreshes and redirects to the same view ID.

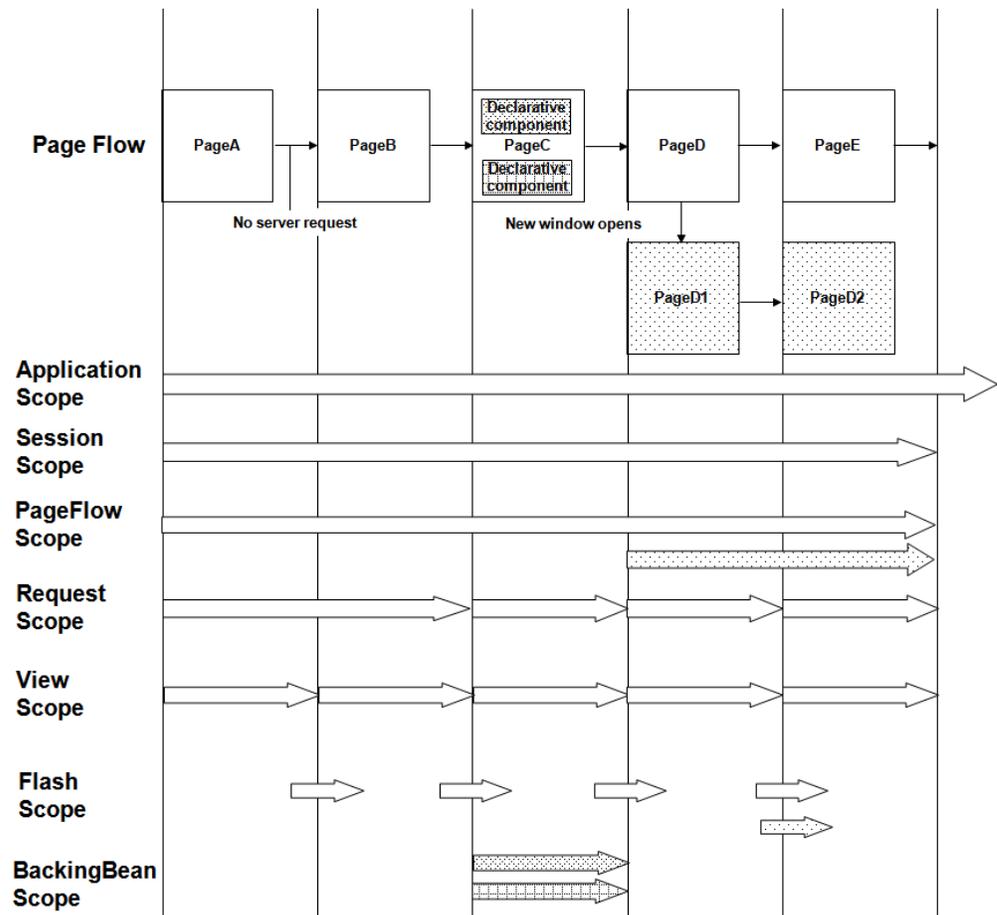
**Note:**

Because these are not standard JSF scopes, EL expressions must explicitly include the scope to reference the bean. For example, to reference the `MyBean` managed bean from the `pageFlowScope` scope, your expression would be `{pageFlowScope.MyBean}`.

Object scopes are analogous to global and local variable scopes in programming languages. The wider the scope, the higher the availability of an object. During their lifespan, these objects may expose certain interfaces, hold information, or pass variables and parameters to other objects. For example, a managed bean defined in `sessionScope` scope will be available for use during multiple page requests. However, a managed bean defined in `requestScope` scope will be available only for the duration of one page request.

[Figure 5-6](#) shows the time period in which each type of scope is valid, and its relationship with the page flow.

Figure 5-6 Relationship Between Scopes and Page Flow



When determining what scope to register a managed bean with or to store a value in, always try to use the narrowest scope possible. Use the `sessionScope` scope only for information that is relevant to the whole session, such as user or context information. Avoid using the `sessionScope` scope to pass values from one page to another.

 **Note:**

If you are using the full Fusion technology stack, then you have the option to register your managed beans in various configuration files. See *Using a Managed Bean in a Fusion Web Application* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Passing Values Between Pages

Oracle ADF provides the `pageFlowScope` bean scope that supports storing temporary values created at runtime. You can use different procedures to access `pageFlowScope` from within any Java code in your application or use `pageFlowScope` without writing any Java code.

The ADF Faces `pageFlowScope` scope makes it easier to pass values from one page to another, thus enabling you to develop master-detail pages more easily. Values added to the `pageFlowScope` scope automatically continue to be available as the user navigates from one page to another, even if you use a `redirect` directive. But unlike `session` scope, these values are visible only in the current page flow or process. If the user opens a new window and starts navigating, that series of windows will have its own process. Values stored in each window remain independent.

 **Note:**

If you are using the full Fusion technology stack and you need information about passing values between pages in an ADF bounded task flow, or between ADF regions and pages, refer to Getting Started with ADF Task Flows in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Like objects stored in any standard JSF scope, objects stored in the `pageFlow` scope can be accessed through EL expressions. The only difference with the `pageFlow` scope is that the object names must use the `pageFlowScope` prefix. For example, to have a button's label provided by a managed bean stored in the `pageFlow` scope, and to have a method on the bean called when the button is selected, you might use the following code on your page:

```
<af:button text="#{pageFlowScope.buttonBean.label}"
           action="#{pageFlowScope.buttonBean.action}"/>
```

The `pageFlowScope` is a `java.util.Map` object that may be accessed from Java code. The `setPropertyListener` tag allows you to set property values onto a scope, and also allows you to define the event the tag should listen for. For example, when you use the `setPropertyListener` tag with the `type` attribute set to `action`, it provides a declarative way to cause an action source (for example, `button`) to set a value before navigation. You can use the `pageFlowScope` scope with the `setPropertyListener` tag to pass values from one page to another, without writing any Java code in a backing bean. For example, you might have one page that uses the `setPropertyListener` tag and a command component to set a value in the `pageFlowScope` scope, and another page whose text components use the `pageFlowScope` scope to retrieve their values.

You can also use the `pageFlowScope` scope to set values between secondary windows such as dialogs. When you launch secondary windows from, for example, a `button` component, you can use a `launchEvent` event and the `pageFlowScope` scope to pass values into and out of the secondary windows without overriding values in the parent process.

How to Use the `pageFlowScope` Scope Within Java Code

You can access `pageFlow` scope from within any Java code in your application. Remember to clear the scope once you are finished.

 **Note:**

If your application uses ADF Controller, then you do not have to manually clear the scope.

Before you begin:

It may be helpful to have an understanding of object scopes. For information, see [Object Scope Lifecycles](#). You may also want to understand how pageFlow scope is used to pass values. For information, see [Passing Values Between Pages](#).

To use pageFlowScope in Java code:

1. To get a reference to the pageFlowScope scope, use the `org.apache.myfaces.trinidad.context.RequestContext.getPageFlowScope()` method.

For example, to retrieve an object from the pageFlowScope scope, you might use the following Java code:

```
import java.util.Map;
import org.apache.myfaces.trinidad.context.RequestContext;
. . .
Map pageFlowScope = RequestContext.getCurrentInstance().getPageFlowScope();
Object myObject = pageFlowScope.get("myObjectName");
```

2. To clear the pageFlowScope scope, access it and then manually clear it.

For example, you might use the following Java code to clear the scope:

```
RequestContext afContext = RequestContext.getCurrentInstance();
afContext.getPageFlowScope().clear();
```

How to Use the pageFlowScope Scope Without Writing Java Code

To use the pageFlowScope scope without writing Java code, use a `setPropertyListener` tag in conjunction with a command component to set a value in the scope. The `setPropertyListener` tag uses the `type` attribute that defines the event type it should listen for. It ignores all events that do not match its type. Once set, you then can access that value from another page within the page flow.

 **Tip:**

Instead of using the `setActionListener` tag (which may have been used in previous versions of ADF Faces), use the `setPropertyListener` tag and set the event type to `action`.

To set a value in the pageFlowScope scope:

1. On the page from where you want to set the value, create a command component using the Components window. For information about creating command components, see [Using Buttons and Links for Navigation](#).

2. In the Components window, from the Listeners group of the Operations panel, drag a **Set Property Listener** and drop it as a child to the command component.
Or right-click the component and choose **Insert inside Button > ADF Faces > setPropertyListener**.
3. In the Insert Set Property Listener dialog, set the **From** field to the value that will be set on another component.
For example, say you have a managed bean named `MyBean` that stores the name value for an employee, and you want to pass that value to the next page. You would enter `#{myBean.empName}` in the **From** field.
4. Set the **To** field to be a value on the `pageFlowScope` scope.
For example, you might enter `#{pageFlowScope.empName}` in the **To** field.
5. From the **Type** dropdown menu, choose **Action**.
This allows the listener to listen for the action event associated with the command component.
6. On the page from which you want to access the value, drop the component that you want to display the value.
7. Set the value of the component to be the same value as the `To` value set on the `setPropertyListener` tag.
For example, to have an `outputText` component access the employee name, you would set the value of that component to be `#{pageFlowScope.empName}`.

What Happens at Runtime: How Values Are Passed

When a user clicks a button that contains a `setPropertyListener` tag, the listener executes and the `To` value is resolved and retrieved, and then stored as a property on the `pageFlowScope` scope. On any subsequent pages that access that property through an EL expression, the expression is resolved to the value set by the original page.

6

Handling Events

This chapter describes how to handle events on the server as well as on the client. This chapter includes the following sections:

- [About Events and Event Handling](#)
- [Using ADF Faces Server Events](#)
- [Using JavaScript for ADF Faces Client Events](#)
- [Sending Custom Events from the Client to the Server](#)
- [Executing a Script Within an Event Response](#)
- [Using ADF Faces Client Behavior Tags](#)
- [Using Polling Events to Update Pages](#)

About Events and Event Handling

ADF Faces supports server-side action and value change events and can also invoke client-side action and value change events. ADF Faces provides a list of event types and event root components.

In traditional JSF applications, event handling typically takes place on the server. JSF event handling is based on the JavaBeans event model, where event classes and event listener interfaces are used by the JSF application to handle events generated by components.

Examples of events in an application include clicking a button or link, selecting an item from a menu or list, and changing a value in an input field. When a user activity occurs such as clicking a button, the component creates an event object that stores information about the event and identifies the component that generated the event. The event is also added to an event queue. At the appropriate time in the JSF lifecycle, JSF tells the component to broadcast the event to the corresponding registered listener, which invokes the listener method that processes the event. The listener method may trigger a change in the user interface, invoke backend application code, or both.

Like standard JSF components, ADF Faces command components deliver `ActionEvent` events when the components are activated, and ADF Faces input and select components deliver `ValueChangeEvent` events when the component local values change.

For example, in the File Explorer application, the File Menu contains a submenu whose `commandMenuItem` components allow a user to create a new file or folder. When users click the **Folder** `commandMenuItem`, an `ActionEvent` is invoked. Because the EL expression set as the value for the component's `actionListener` attribute resolves to the `createNewDirectory` method on the `headerManager` managed bean, that method is invoked and a new directory is created.

**Note:**

Any ADF Faces component that has built-in event functionality must be enclosed in the `form` tag.

While ADF Faces adheres to standard JSF event handling techniques, it also enhances event handling in two key ways by providing:

- Ajax-based functionality (partial page rendering)
- A client-side event model

Events and Partial Page Rendering

Unlike standard JSF events, ADF Faces events support Ajax-style partial postbacks to enable partial page rendering (PPR). Instead of full page rendering, ADF Faces events and components can trigger partial page rendering, that is, only portions of a page refresh upon request and the lifecycle is run only on that portion.

Certain components are considered **event root** components. Event root components determine boundaries on the page, and so allow the lifecycle to run just on components within that boundary (for information about this aspect of the lifecycle, see [Using the Optimized Lifecycle](#)). When an event occurs within an event root, only those components that are children to the root are refreshed on the page.

An example of an event root component is a popup. When an event happens within a popup, only the popup and its children are rerendered, and not the whole page. Another example is the `panelCollection` component that surrounds a table and provides among other things, a toolbar. Action events triggered by a button on the toolbar will cause the lifecycle to be run only on the child components of the `panelCollection` component.

The following components are considered event root components:

- `popup`
- `region`
- `panelCollection`
- `calendar`
- `editableValueHolder` components (such as `inputText`)

Additionally, certain events indicate a specific component as an event root component. For example, the disclosure event sent when a expanding or collapsing a `showDetail` component (see [Displaying and Hiding Contents Dynamically](#)), indicates that the `showDetail` component is a root. The lifecycle is run only on the `showDetail` component (and any child components or other components that point to this as a trigger), and only they are rerendered when it is expanded or collapsed. For information about components and their associated events, see [Event and Even Root Components](#).

 **Tip:**

If components outside of the event root need to be processed when the event root is processed, then you can programmatically determine which components should participate, and whether they should be executed in the lifecycle or simply rendered. See [Rerendering Partial Page Content](#).

Event and Event Root Components

[Table 6-1](#) shows the all event types in ADF Faces, and whether or not the source component is an event root.

Table 6-1 Events and Event Root Components

Event Type	Component Trigger	Is Event Root
action	All command components	false
dialog	dialog	false
disclosure	showDetail, showDetailHeader	true
disclosure	showDetailItem	true
focus	tree, treeTable	true
launch	All command components	NA
launchPopup	inputListOfValues, inputComboboxListOfValues	true
load	document	NA
poll	poll	true
popupOpened	popup	NA
popupOpening	popup	NA
popupClosed	popup	NA
propertyChange	All components	NA
queryEvent	query, quickQuery	true
queryOperation	query, quickQuery	true
rangeChange	table	NA
regionNavigation	region	NA
return	All command components	true
returnPopupData	inputListOfValues, inputComboboxListOfValues	true
returnPopup	inputListOfValues, inputComboboxListOfValues	true
rowDisclosure	tree, treeTable, treemap, sunburst	true
sort	treeTable, table	true

Table 6-1 (Cont.) Events and Event Root Components

Event Type	Component Trigger	Is Event Root
valueChange	All input and select components (components that implement <code>EditableValueHolder</code>)	true

Client-Side Event Model

In addition to server-side action and value change events, ADF Faces components also invoke client-side action and value change events, and other kinds of server and client events. Some events are generated by both server and client components (for example, selection events); some events are generated by server components only (for example, launch events); and some events are generated by client components only (for example, load events).

By default, most client events are propagated to the server. Changes to the component state are automatically synchronized back to the server to ensure consistency of state, and events are delivered, when necessary, to the server for further processing. However, you can configure your event so that it does not propagate.

In addition, any time you register a client-side event listener on the server-side Java component, the ADF Faces framework assumes that you require a JavaScript component, so a client-side component is created.

Client-side JavaScript events can come from several sources: they can be derived automatically from DOM events, from property change events, or they can be manually created during the processing of other events.

Using ADF Faces Server Events

ADF Faces supports server-side action and value change events. To process the events, you must understand the different server-side event types available and the components that trigger these events.

ADF Faces provides a number of server-side events. [Table 6-2](#) lists the events generated by ADF Faces components on the server, and the components that trigger them.

Table 6-2 ADF Faces Server Events

Event	Triggered by Component...
ActionEvent	All command components. See Working with Navigation Components .
ActiveDataEvent	Used to update components based on events. See <i>Using the Active Data Service in Developing Fusion Web Applications with Oracle Application Development Framework</i> .
AttributeChangeEvent	All input and select components (components that implement <code>EditableValueHolder</code>). See Using Input Components and Defining Forms .

Table 6-2 (Cont.) ADF Faces Server Events

Event	Triggered by Component...
CalendarActivity DurationChangeEvent CalendarActivityEvent CalendarDisplay ChangeEvent CalendarEvent	The Calendar component. See Using a Calendar Component .
CarouselSpinEvent	The carousel component. See Displaying Images in a Carousel .
ColumnSelectionEvent ColumnVisibility ChangeEvent	The table and treeTable components. See Using Tables, Trees, and Other Collection-Based Components .
ContextInfoEvent	The contextInfo component. See Displaying Contextual Information in Popups .
DialogEvent	The dialog component. See Using Popup Dialogs, Menus, and Windows .
DisclosureEvent	The showDetail, showDetailHeader, showDetailItem components. See Displaying and Hiding Contents Dynamically and Displaying or Hiding Contents in Panels .
DropEvent	Components that support drag and drop. See Adding Drag and Drop Functionality .
FocusEvent *	The tree and treeTable components. See Using Tables, Trees, and Other Collection-Based Components .
ItemEvent	The panelTabbed component. See Displaying or Hiding Contents in Panels . Also, the navigationPane component. See Using Navigation Items for a Page Hierarchy .
LaunchEvent	All command components. See Working with Navigation Components .
LaunchPopupEvent	The inputListOfValues and inputComboboxListOfValues components. See Using Query Components .
LoadEvent **	The document component. See How to Configure the document Tag .
PollEvent	The poll component. See Using Polling Events to Update Pages .
PopupCanceledEvent PopupFetchEvent	The popup component. See Using Popup Dialogs, Menus, and Windows .
QueryEvent QueryOperationEvent	The query and quickQuery components. See Using Query Components .
RangeChangeEvent	The table component. See Using Tables, Trees, and Other Collection-Based Components .
RegionNavigationEvent	The region component. See Using Task Flows as Regions in Developing Fusion Web Applications with Oracle Application Development Framework .

Table 6-2 (Cont.) ADF Faces Server Events

Event	Triggered by Component...
ReturnEvent	All command components. See Working with Navigation Components .
ReturnPopupEvent	The inputListOfValues and inputComboboxListOfValues components. See Using Query Components .
ReturnPopupDataEvent	The popup component. See Using Popup Dialogs, Menus, and Windows .
RowDisclosureEvent	The tree and treeTable components, as well as the treemap and sunburst DVT components. See Using Tables, Trees, and Other Collection-Based Components and Using Treemap and Sunburst Components .
SelectionEvent	The table, tree, and treeTable components, as well as the treemap and sunburst DVT components. See Using Tables, Trees, and Other Collection-Based Components and Using Treemap and Sunburst Components .
SortEvent	The table and treeTable components. See Using Tables, Trees, and Other Collection-Based Components .
ValueChangeEvent	All input and select components (components that implement EditableValueHolder). See Using Input Components and Defining Forms .
WindowLifecycleEvent	Delivered when the LifecycleState of a window changes. See the <i>Java API Reference for Oracle ADF Faces</i> .
WindowLifecycleNavigateEvent	Delivered when the current window is unloaded in order to navigate to a new location. See the <i>Java API Reference for Oracle ADF Faces</i> .

* This focus event is generated when focusing in on a specific subtree, which is not the same as a client-side keyboard focus event.

** The LoadEvent event is fired after the initial page is displayed (data streaming results may arrive later).

How to Handle Server-Side Events

All server events have event listeners on the associated component(s). You need to create a handler that processes the event and then associate that handler code with the listener on the component.

For example, in the File Explorer application, a selection event is fired when a user selects a row in the table. Because the table's selectionListener attribute is bound to the tableSelectFileItem handler method on the TableContentView.java managed bean, that method is invoked in response to the event.

Before you begin:

It may be helpful to have an understanding of server-side events. See [Using ADF Faces Server Events](#).

To handle server-side events:

1. In a managed bean (or the backing bean for the page that will use the event listener), create a public method that accepts the event (as the event type) as the only parameter and returns `void`. The following example shows the code for the `tableSelectFileItem` handler. (For information about creating and using managed beans, see [Creating and Using Managed Beans](#).)

```
public void tableSelectFileItem(SelectionEvent selectionEvent)
{
    FileItem data = (FileItem)this.getContentTable().getSelectedRowData();
    setSelectedFileItem(data);
}
```

 **Tip:**

If the event listener code is likely to be used by more than one page in your application, consider creating an event listener implementation class that all pages can access. All server event listener class implementations must override a `processEvent()` method, where `Event` is the event type.

For example, the `LaunchListener` event listener accepts an instance of `LaunchEvent` as the single argument. In an implementation, you must override the event processing method, as shown in the following method signature:

```
public void processLaunch (LaunchEvent evt)
{
    // your code here
}
```

2. To register an event listener method on a component, in the Structure window, select the component that will invoke the event. In the Properties window, use the dropdown menu next to the event listener property, and choose **Edit**.
3. Use the Edit Property dialog to select the managed bean and method created in Step 1.

The following example shows sample code for registering a selection event listener method on a `table` component.

```
<af:table id="folderTable" var="file"
. . .
        rowSelection="single"
        selectionListener="#{explorer.tableContentView.tableSelectFileItem}"
. . .
</af:table>
```

Using JavaScript for ADF Faces Client Events

ADF Faces can invoke client-side action and value change events. To process client-side events, you must understand the different event types available and also the components that trigger these events. To use client-side events, you need to first create the JavaScript that will handle the event.

Most components can also work with client-side events. Handling events on the client saves a roundtrip to the server. When you use client-side events, instead of having

managed beans contain the event handler code, you use JavaScript, which can be contained either on the calling page or in a JavaScript library.

By default, client events are processed only on the client. However, some event types are also delivered to the server, for example, `AdfActionEvent` events, which indicate a button has been clicked. Other events may be delivered to the server depending on the component state. For example, `AdfValueChangeEvent` events will be delivered to the server when the `autoSubmit` attribute is set to `true`. You can cancel an event from being delivered to the server if no additional processing is needed. However, some client events cannot be canceled. For example, because the `popupOpened` event type is delivered after the popup window has opened, this event delivery to the server cannot be canceled.

Performance Tip:

If no server processing is needed for an event, consider canceling the event at the end of processing so that the event does not propagate to the server. See [How to Prevent Events from Propagating to the Server](#).

Best Practice:

Keyboard and mouse events wrap native DOM events using the `AdfUIInputEvent` subclass of the `AdfBaseEvent` class, which provides access to the original DOM event and also offers a range of convenience functions for retrieval of key codes, mouse coordinates, and so on. The `AdfBaseEvent` class also accounts for browser differences in how these events are implemented. Consequently, you must avoid invoking the `getNativeEvent()` method on the directly, and instead use the `AdfUIInputEvent` API.

The `clientListener` tag provides a declarative way to register a client-side event handler script on a component. The script will be invoked when a supported client event type is fired. The following shows an example of a JavaScript function associated with an action event.

```
<af:button id="button0"
           text="Do something in response to an action">
  <af:clientListener method="someJSMethod" type="action"/>
</af:button>
```

Tip:

Use the `clientListener` tag instead of the component's JavaScript event properties.

All ADF Faces components support the JSF 2.0 client behavior API. Client events on ADF Faces components are also exposed as client behaviors. Client behaviors tags (like `f:ajax`) allow you to declaratively attach JavaScript to a component, which will

then execute in response to a client behavior. For example, the following code shows the `f:ajax` tag attached to an `inputText` component. This tag will cause the `outputText` component to render when the `change` client event occurs on the `inputText` component.

```
af:inputText ...>
  <f:ajax name="change" render="ot1" execute="@this" />
</af:inputText>
<af:outputText id="ot1" ... />
```

ADF Faces Client-Side Events

[Table 6-3](#) lists the events generated by ADF Faces client components, whether or not events are sent to the sever, whether or not the events are cancelable, and the components that trigger the events.

Table 6-3 ADF Faces Client Events

Event Class	Event Type	Propagates to Server	Can Be Canceled	Triggered by Component
<code>AdfActionEvent</code>	<code>action</code>	Yes	Yes	All command components
<code>AdfBusyStateEvent</code>	<code>busyState</code>	No	No	Triggered by the page
<code>AdfCarouselSpinEvent</code>	<code>event</code>	Yes	No	<code>carousel</code>
<code>AdfChooseDateLoadEvent</code>	<code>load</code>	No	Yes	<code>chooseDate</code>
<code>AdfColumnSelectionEvent</code>	<code>event</code>	Yes	Yes	<code>table</code> , <code>treeTable</code>
<code>AdfComponentEvent</code>	<code>load</code>	Yes	Yes	<code>document</code> After the document's contents have been displayed on the client, even when PPR navigation is used. It does not always correspond to the <code>onLoad</code> DOM event.
<code>AdfComponentFocusEvent</code>		No	Yes	Any component that can receive focus
<code>AdfDateSelectionEvent</code>	<code>dateSelection</code>	No	Yes	<code>chooseDate</code>
<code>AdfDialogEvent</code>	<code>event</code>	Yes	Yes	<code>dialog</code> When user selects the OK or Cancel button in a dialog
<code>AdfDisclosureEvent</code>	<code>event</code>	Yes	Yes	<code>panelBox</code> , <code>region</code> , <code>showDetail</code> , <code>showDetailHeader</code> , <code>showDetailItem</code> When the disclosure state is toggled by the user
<code>AdfDomComponentEvent</code>	<code>inlineFrameLoad</code>	Yes	Yes	<code>inlineFrame</code> When the internal <code>iframe</code> fires its load event.
<code>AdfDropEvent</code>	<code>drop</code>	Yes	No	Any component that supports drag and drop

Table 6-3 (Cont.) ADF Faces Client Events

Event Class	Event Type	Propagates to Server	Can Be Canceled	Triggered by Component
AdfFocusEvent	focus	Yes	Yes	tree, treeTable
AdfItemEvent	item	Yes	Yes	commandNavigationItemshow DetailItem
AdfLaunchPopupEvent	launch	Yes	Yes	inputListOfValues, inputComboboxListOfValues
AdfPollEvent	poll	Yes	Yes	poll
AdfPopupCanceledEvent	popupCanceled	Yes	Yes	popup After a popup is unexpectedly closed or the cancel method is invoked
AdfPopupClosedEvent	popupClosed	No	No	popup After a popup window or dialog is closed
AdfPopupOpenedEvent	popupOpened	No	No	popup After a popup window or dialog is opened
AdfPopupOpeningEvent	popupOpening	No	Yes	popup Prior to opening a popup window or dialog
AdfPropertyChangeEvent	propertyChange	No	No	All components
AdfQueryEvent	event	Yes	Yes	query, quickQuery Upon a query action (that is, when the user clicks the search icon or search button)
AdfQueryOperationEvent	event	Yes	Yes	query, quickQuery
AdfReturnEvent	returnEvent	Yes	Yes	All command components
AdfReturnPopupDataEvent	launchEvent	Yes	Yes	inputListOfValues, inputComboboxListOfValues
AdfReturnPopupEvent	returnPopup	Yes	Yes	inputListOfValues, inputComboboxListOfValues
AdfRowDisclosureEvent	rowDisclosure	Yes	Yes	tree, treeTable When the row disclosure state is toggled
AdfRowKeySetChangeEvent	selection, rowDisclosure	Always for disclosure event on a table. Yes, if there is a selection listener or a disclosure listener on the server.	Yes	table, treeTable, tree

Table 6-3 (Cont.) ADF Faces Client Events

Event Class	Event Type	Propagates to Server	Can Be Canceled	Triggered by Component
AdfSelectionEvent	selection	Yes	Yes	tree, treeTable, table When the selection state changes
AdfSortEvent	sort	Yes	Yes	treeTable, table When the user sorts the table data
AdfValueChangeEvent	valueChange	Yes	Yes	All input and select components (components that implement EditableValueHolder) When the value of an input or select component is changed

ADF Faces also supports client keyboard and mouse events, as shown in [Table 6-4](#).

Table 6-4 Keyboard and Mouse Event Types Supported

Event Type	Event Fires When...
click	User clicks a component
dblclick	User double-clicks a component
mousedown	User moves mouse down on a component
mouseup	User moves mouse up on a component
mousemove	User moves mouse while over a component
mouseover	Mouse enters a component
mouseout	Mouse leaves a component
keydown	User presses key down while focused on a component
keyup	User releases key while focused on a component
keypress	When a successful keypress occurs while focused on a component
focus	Component gains keyboard focus
blur	Component loses keyboard focus

How to Use Client-Side Events

To use client-side events, you need to first create the JavaScript that will handle the event. You then use a `clientListener` tag.

Before you begin:

It may be helpful to have an understanding of client-side events. See [Using JavaScript for ADF Faces Client Events](#).

To use client-side events:

1. Create the JavaScript event handler function. For information about creating JavaScript, see [Adding JavaScript to a Page](#). Within that functionality, you can add the following:
 - Locate a client component on a page

If you want your event handler to operate on another component, you must locate that component on the page. For example, in the File Explorer application, when users choose the **Give Feedback** menu item in the **Help** menu, the associated JavaScript function has to locate the help popup dialog in order to open it. For information about locating client components, see [Locating a Client Component on a Page](#).
 - Return the original source of the event

If you have more than one of the same component on the page, your JavaScript function may need to determine which component issued the event. For example, say more than one component can open the same popup dialog, and you want that dialog aligned with the component that called it. You must know the source of the `AdfLaunchPopupEvent` in order to determine where to align the popup dialog. See [How to Return the Original Source of the Event](#).
 - Add client attributes

It may be that your client event handler will need to work with certain attributes of a component. For example, in the File Explorer application, when users choose the **About** menu item in the **Help** menu, a dialog launches that allows users to provide feedback. The function used to open and display this dialog is also used by other dialogs, which may need to be displayed differently. Therefore, the function needs to know which dialog to display along with information about how to align the dialog. This information is carried in client attributes. Client attributes can also be used to marshal custom server-side attributes to the client. See [How to Use Client-Side Attributes for an Event](#).
 - Cancel propagation to the server

Some of the components propagate client-side events to the server, as shown in [Table 6-3](#). If you do not need this extra processing, then you can cancel that propagation. See [How to Prevent Events from Propagating to the Server](#).
2. Once you create the JavaScript function, you must add an event listener that will call the event method.

 **Note:**

Alternatively, you can use a JSF 2.0 client behavior tag (such as `f:ajax`) to respond to the client event, as all client events on ADF Faces components are also exposed as client behaviors. See the Java EE 6 tutorial (<http://download.oracle.com/javaee/index.html>)

- a. Select the component to invoke the JavaScript, and in the Properties window, set **ClientComponent** to **true**.
- b. In the Components window, from the Operations panel, in the Listeners group, drag a **Client Listener** and drop it as a child to the selected component.

- c. In the Insert Client Listener dialog, enter the method and select the type for the JavaScript function.

The `method` attribute of the `clientListener` tag specifies the JavaScript function to call when the corresponding event is fired. The JavaScript function must take a single parameter, which is the event object.

The `type` attribute of the `clientListener` tag specifies the client event type that the tag will listen for, such as `action` or `valueChange`. Table 6-3 lists the ADF Faces client events.

The `type` attribute of the `clientListener` tag also supports client event types related to keyboard and mouse events. Table 6-4 lists the keyboard and mouse event types.

The following example shows the code used to invoke the `showHelpFileExplorerPopup` function from the `Explorer.js` JavaScript file.

```
<af:commandMenuItem id="feedbackMenuItem"
    text="#{explorerBundle['menuItem.feedback']}"
    clientComponent="true">
    <af:clientListener method="Explorer.showHelpFileExplorerPopup"
        type="action"/>
</af:commandMenuItem>
```

- d. To add any attributes required by the function, in the Components window, from the Operations panel, drag a **Client Attribute** and drop it as a child to the selected component. Enter the name and value for the attribute in the Properties window. The following example shows the code used to set attribute values for the `showAboutFileExplorerPopup` function.

```
<af:commandMenuItem id="aboutMenuItem"
    text="#{explorerBundle['menuItem.about']}"
    clientComponent="true">
    <af:clientListener method="Explorer.showAboutFileExplorerPopup"
        type="action"/>
    <af:clientAttribute name="popupCompId" value=":fe:aboutPopup"/>
    <af:clientAttribute name="align" value="end_after"/>
    <af:clientAttribute name="alignId" value="aboutMenuItem"/>
</af:commandMenuItem>
```

Best Practice:

Keyboard and mouse events wrap native DOM events using the `AdfUIInputEvent` subclass of the `AdfBaseEvent` class, which provides access to the original DOM event and also offers a range of convenience functions for retrieval of key codes, mouse coordinates, and so on. The `AdfBaseEvent` class also accounts for browser differences in how these events are implemented. Consequently, you must avoid invoking the `getNativeEvent()` method on the directly, and instead use the `AdfUIInputEvent` API.

How to Return the Original Source of the Event

The JavaScript method `getSource()` returns the original source of a client event. For example, the File Explorer application contains the `showAboutFileExplorerPopup` function shown in the following example, that could be used by multiple events to set

the alignment on a given popup dialog or window, using client attributes to pass in the values. Because each event that uses the function may have different values for the attributes, the function must know which source fired the event so that it can access the corresponding attribute values (for more about using client attributes, see [How to Use Client-Side Attributes for an Event](#)).

```
Explorer.showAboutFileExplorerPopup = function(event)
{
    var source = event.getSource();
    var alignType = source.getProperty("align");
    var alignCompId = source.getProperty("alignId");
    var popupCompId = source.getProperty("popupCompId");

    source.show({align:alignType, alignId:alignCompId});

    event.cancel();
}
```

The `getSource()` method is called to determine the client component that fired the current focus event, which in this case is the popup component.

How to Use Client-Side Attributes for an Event

There may be cases when you want the script logic to cause some sort of change on a component. To do this, you may need attribute values passed in by the event. For example, the File Explorer application contains the `showAboutFileExplorerPopup` function shown in the following example, that can be used to set the alignment on a given popup component, using client attributes to pass in the values. The attribute values are accessed by calling the `getProperty` method on the source component.

```
Explorer.showAboutFileExplorerPopup = function(event)
{
    var source = event.getSource();
    var alignType = source.getProperty("align");
    var alignCompId = source.getProperty("alignId");
    var popupCompId = source.getProperty("popupCompId");

    var aboutPopup = event.getSource().findComponent(popupCompId);
    aboutPopup.show({align:alignType, alignId:alignCompId});

    event.cancel();
}
```

The values are set on the source component, as shown in the following example.

```
<af:commandMenuItem id="aboutMenuItem"
    text="{explorerBundle['menuItem.about']}"
    clientComponent="true">
    <af:clientListener method="Explorer.showAboutFileExplorerPopup"
        type="action"/>
    <af:clientAttribute name="popupCompId" value=":aboutPopup"/>
    <af:clientAttribute name="align" value="end_after"/>
    <af:clientAttribute name="alignId" value="aboutMenuItem"/>
</af:commandMenuItem>
```

Using attributes in this way allows you to reuse the script across different components, as long as they all trigger the same event.

How to Block UI Input During Event Execution

There may be times when you do not want the user to be able to interact with the UI while a long-running event is processing. For example, suppose your application uses a button to submit an order, and part of the processing includes creating a charge to the user's account. If the user were to inadvertently press the button twice, the account would be charged twice. By blocking user interaction until server processing is complete, you ensure no erroneous client activity can take place.

The ADF Faces JavaScript API includes the `AdfBaseEvent.preventUserInput` function. To prevent all user input while the event is processing, you can call the `preventUserInput` function, and a glass pane will cover the entire browser window, preventing further input until the event has completed a roundtrip to the server.

You can use the `preventUserInput` function only with custom events, events raised in a custom client script, or events raised in a custom client component's peer. Additionally, the event must propagate to the server. The following example shows how you can use `preventUserInput` in your JavaScript.

```
function queueEvent(event)
{
    event.cancel(); // cancel action event
    var source = event.getSource();

    var params = {};
    var type = "customListener";
    var immediate = true;
    var isPartial = true;
    var customEvent = new AdfCustomEvent(source, type, params, immediate);
    customEvent.preventUserInput();
    customEvent.queue(isPartial);
}
```

How to Prevent Events from Propagating to the Server

By default, some client events propagate to the server once processing has completed on the client. In some circumstances, it is desirable to block this propagation. For instance, if you are using a `button` component to execute JavaScript code when the button is clicked, and there is no `actionListener` event listener on the server, propagation of the event is a waste of resources. To block propagation to the server, you call the `cancel()` function on the event in your listener. Once the `cancel()` function has been called, the `isCanceled()` function will return `true`.

The following example shows the `showAboutFileExplorerPopup` function, which cancels its propagation.

```
Explorer.showAboutFileExplorerPopup = function(event)
{
    var source = event.getSource();
    var alignType = source.getProperty("align");
    var alignCompId = source.getProperty("alignId");
    var popupCompId = source.getProperty("popupCompId");

    var aboutPopup = event.getSource().findComponent(popupCompId);
    aboutPopup.show({align:alignType, alignId:alignCompId});
}
```

```
    event.cancel();  
}
```

Canceling an event may also block some default processing. For example, canceling an `AdfUIInputEvent` event for a context menu will block the browser from showing a context menu in response to that event.

The `cancel()` function call will be ignored if the event cannot be canceled, which an event indicates by returning `false` from the `isCancelable()` function (events that cannot be canceled show "no" in the Is Cancelable column in [Table 6-3](#)). This generally means that the event is a notification that an outcome has already completed, and cannot be blocked. There is also no way to uncanceled an event once it has been canceled.

How to Indicate No Response is Expected

There may be times when you do not expect the framework to handle the response for an event. For example, when exporting table content to a spreadsheet, you don't need to wait for the call to return. To let the framework know that no response is expected, you use the `AdfBaseEvent.noResponseExpected()` method.

What Happens at Runtime: How Client-Side Events Work

Event processing in general is taken from the browser's native event loop. The page receives all DOM events that bubble up to the document, and hands them to the peer associated with that piece of DOM. The peer is responsible for creating a JavaScript event object that wraps that DOM event, returning it to the page, which queues the event (for information about peers and the ADF Faces architecture, see [Using ADF Faces Client-Side Architecture](#)).

The event queue on the page most commonly empties at the end of the browser's event loop once each DOM event has been processed by the page (typically, resulting in a component event being queued). However, because it is possible for events to be queued independently of any user input (for example, poll components firing their poll event when a timer is invoked), queueing an event also starts a timer that will force the event queue to empty even if no user input occurs.

The event queue is a First-In-First-Out queue. For the event queue to empty, the page takes each event object and delivers it to a `broadcast()` function on the event source. This loop continues until the queue is empty. It is completely legitimate (and common) for broadcasting an event to indirectly lead to queueing a new, derived event. That derived event will be broadcast in the same loop.

When an event is broadcast to a component, the component does the following:

1. Delivers the event to the peer's `DispatchComponentEvent` method.
2. Delivers the event to any listeners registered for that event type.
3. Checks if the event should be bubbled, and if so initiates bubbling. Most events do bubble. Exceptions include property change events (which are not queued, and do not participate in this process at all) and, for efficiency, mouse move events.

While an event is bubbling, it is delivered to the `AdfUIComponent.HandleBubbledEvent` function, which offers up the event to the peer's `DispatchComponentEvent` function. Note that client event listeners do not receive the event, only the peers do.

Event bubbling can be blocked by calling an event's `stopBubbling()` function, after which the `isBubblingStopped()` function will return `true`, and bubbling will not continue. As with cancelling, you cannot undo this call.

 **Note:**

Cancelling an event does not stop bubbling. If you want to both cancel an event and stop it from bubbling, you must call both functions.

4. If none of the prior work has canceled the event, calls the `AdfUIComponent.HandleEvent` method, which adds the event to the server event queue, if the event requests it.

What You May Need to Know About Using Naming Containers

Several components in ADF Faces are `NamingContainer` components, such as `pageTemplate`, `subform`, `table`, and `tree`. When working with client-side API and events in pages that contain `NamingContainer` components, you should use the `findComponent()` method on the source component.

For example, because all components in any page within the File Explorer application eventually reside inside a `pageTemplate` component, any JavaScript function must use the `getSource()` and `findComponent()` methods, as shown in the following example. The `getSource()` method accesses the `AdfUIComponent` class, which can then be used to find the component.

```
function showPopup(event)
{
    event.cancel();
    var source = event.getSource();
    var popup = source.findComponent("popup");
    popup.show({align:"after_end", alignId:"button"});
}
```

When you use the `findComponent()` method, the search starts locally at the component where the method is invoked. For information about working with naming containers, see [Locating a Client Component on a Page](#).

Sending Custom Events from the Client to the Server

In ADF Faces, you can use a custom event to send any custom data back to the server from the client. To send a custom event from the client to the server, fire the client event using a custom event type, write the server listener method on a backing bean, and have this method process the custom event, and then register the server listener with the component.

While the `clientAttribute` tag supports sending bonus attributes from the server to the client, those attributes are not synchronized back to the server. To send any custom data back to the server, use a custom event sent through the `AdfCustomEvent` class and the `serverListener` tag.

The `AdfCustomEvent.queue()` JavaScript method enables you to fire a custom event from any component whose `clientComponent` attribute is set to `true`. The custom event object contains information about the client event source and a map of

parameters to include on the event. The custom event can be set for immediate delivery (that is, during the Apply Request Values phase), or non-immediate delivery (that is, during the Invoke Application phase).

For example, in the File Explorer application, after entering a file name in the search field on the left, users can press the Enter key to invoke the search. As the following example shows, this happens because the `inputText` field contains a `clientListener` that invokes a JavaScript function when the **Enter** key is pressed.

```
//Code on the JSF page...
<af:inputText id="searchCriteriaName"
    value="#{explorer.navigatorManager.searchNavigator.
        searchCriteriaName}"
    shortDesc="#{explorerBundle['navigator.filenamesearch']}">
<af:serverListener type="enterPressedOnSearch"
    method="#{explorer.navigatorManager.
        searchNavigator.searchOnEnter}"/>
<af:clientListener type="keyPress"
    method="Explorer.searchNameHandleKeyPress"/>
</af:inputText>

//Code in JavaScript file...
Explorer.searchNameHandleKeyPress = function (event)
{
    if (event.getKeyCode()==AdfKeyStroke.ENTER_KEY)
    {
        var source = event.getSource();
        AdfCustomEvent.queue(source,
            "enterPressedOnSearch",
            {},
            false);
    }
}
```

The JavaScript contains the `AdfCustomEvent.queue` method that takes the event source, the string `enterPressedOnSearch` as the custom event type, a null parameter map, and `False` for the immediate parameter.

The `inputText` component on the page also contains the following `serverListener` tag:

```
<af:serverListener type="enterPressedOnSearch"
    method="#{explorer.navigatorManager.
        searchNavigator.searchOnEnter}"/>
```

Because the type value `enterPressedOnSearch` is the same as the value of the parameter in the `AdfCustomEvent.queue` method in the JavaScript, the method that resolves to the method expression `#{explorer.navigatorManager.searchNavigator.searchOnEnter}` will be invoked.

How to Send Custom Events from the Client to the Server

To send a custom event from the client to the server, fire the client event using a custom event type, write the server listener method on a backing bean, and have this method process the custom event. Next, register the server listener with the component.

Before you begin:

It may be helpful to have an understanding of sending custom events to the server. See [Sending Custom Events from the Client to the Server](#).

To send custom events:

1. Create the JavaScript that will handle the custom event using the `AdfCustomEvent.queue()` method to provide the event source, custom event type, and the parameters to send to the server.

For example, the JavaScript used to cause the pressing of the Enter key to invoke the search functionality uses the `AdfCustomEvent.queue` method that takes the event source, the string `enterPressedOnSearch` as the custom event type, a null parameter map, and `False` for the immediate parameter, as shown in the following example.

```
Explorer.searchNameHandleKeyPress = function (event)
{
  if (event.getKeyCode()==AdfKeyStroke.ENTER_KEY)
  {
    var source = event.getSource();
    AdfCustomEvent.queue(source,
                        "enterPressedOnSearch",
                        {},
                        false);
  }
}
```

2. Create the server listener method on a managed bean. This method must be public and take an `oracle.adf.view.rich.render.ClientEvent` object and return a void type. The following example shows the code used in the `SearchNavigatorView` managed bean that simply calls another method to execute the search and then refreshes the navigator.

```
public void searchOnEnter(ClientEvent clientEvent)
{
  doRealSearchForFileItem();

  // refresh search navigator
  this.refresh();
}
```

 **Note:**

The Java-to-JavaScript transformation can lose type information for Numbers, chars, Java Objects, arrays, and nonstring CharSequences. Therefore, if an object being sent to the server was initially on the server, you may want to add logic to ensure the correct conversion. See [What You May Need to Know About Marshalling and Unmarshalling Data](#).

3. To register the `clientListener`, in the Components window, from the Operations panel, drag a **Client Listener** and drop it as a child to the component that raises the event.

 **Note:**

On the component that will fire the custom client event, the `clientComponent` attribute must be set to `true` to ensure that a client-side generated component is available.

4. In the Insert Client Listener dialog, enter the method and type for the JavaScript function. Be sure to include a library name if the script is not included on the page. The type can be any string used to identify the custom event, for example, `enterPressedOnSearch` was used in the File Explorer.
5. To register the server listener, in the Components window, from the Operations panel, drag a **Server Listener** and drop it as a sibling to the `clientListener` tag.
6. In the Insert Server Listener dialog, enter the string used as the Type value for the client listener, as the value for this server listener, for example `enterPressedOnSearch`.

In the Properties window, for the `method` attribute, enter an expression that resolves to the method created in Step 2.

What Happens at Runtime: How Client and Server Listeners Work Together

At runtime, when the user initiates the event, for example, pressing the Enter key, the client listener script executes. This script calls the `AdfCustomEvent.queue()` method, and a custom event of the specified event type is queued on the input component. The server listener registered on the input component receives the custom event, and the associated bean method executes.

What You May Need to Know About Marshalling and Unmarshalling Data

Marshalling and unmarshalling is the process of converting data objects of a programming language into a byte stream and back into data objects that are native to the same or a different programming language. In ADF Faces, marshalling and unmarshalling refer to transformation of data into a suitable format so that it can be optimally exchanged between JavaScript on the client end and Java on the server end.

Note that there could be some loss of information during the conversion process. For example, say you are using the following custom event to send the number 1 and the String `test`, as shown in the following example:

```
AdfCustomEvent.queue(event.getSource(), "something", {first:1, second:"test"});
```

In the server-side listener, the type of the `first` parameter would become a `java.lang.Double` because numbers are converted to `Doubles` when going from JavaScript to Java. However, it might be that the parameter started on the server side as an `int`, and was converted to a number when conversion from Java to JavaScript took place. Now on its return trip to the server, it will be converted to a `Double`.

[Mapping Java to JavaScript](#) provides mapping between Java and JavaScript types.

Mapping Java to JavaScript

Table 6-5 shows how JavaScript types are mapped to the corresponding Java types in ADF Faces.

Table 6-5 JavaScript to Java Type Map

JavaScript Type	Java Type
Boolean	<code>java.lang.Boolean</code>
Number	<code>java.lang.Double</code>
String	<code>java.lang.String</code>
Date	<code>java.util.Date</code>
Array	<code>java.util.ArrayList</code>
Object	<code>java.util.Map</code>

Table 6-6 shows how Java types map back to JavaScript types.

Table 6-6 Java to JavaScript Type Map

Java Type	JavaScript Type
<code>java.lang.Boolean</code>	Boolean
<code>java.lang.Double</code>	Number
<code>java.lang.Integer</code>	Number
<code>java.lang.Float</code>	Number
<code>java.lang.Long</code>	Number
<code>java.lang.Short</code>	Number
<code>java.lang.Character</code>	String
<code>java.lang.CharSequence</code>	String
<code>java.util.Collection</code>	Array
<code>java.util.Date</code>	Date
<code>java.util.Map</code>	Object
Array	Array
<code>java.awt.Color</code>	TrColor

Executing a Script Within an Event Response

Executing JavaScript within an event response is useful when you have a JavaScript library that does certain operations on the client side, and you want to call these functions after finishing an action on the server side.

Using the `ExtendedRenderKitService` class, you can add JavaScript to an event response, for example, after invoking an action method binding. It can be a simple message like sending an alert informing the user that the database connection could

not be established, or a call to a function like `hide()` on a popup window to programatically dismiss a popup dialog.

For example, in the File Explorer application, when the user clicks the `UpOneFolder` navigation button to move up in the folder structure, the folder pane is repainted to display the parent folder as selected. The `HandleUpOneFolder()` method is called in response to clicking the `UpOneFolder` button event. It uses the `ExtendedRenderKitService` class to add JavaScript to the response.

The following example shows the `UpOneFolder` code in the page with the `actionListener` attribute bound to the `HandleUpOneFolder()` handler method which will process the action event when the button is clicked.

```
<af:btton id="upOneFolder"
. . .
    actionListener="#{explorer.headerManager.handleUpOneFolder}"/>
```

The following example shows the `handleUpOneFolder` method that uses the `ExtendedRenderKitService` class.

```
public void handleUpOneFolder(ActionEvent actionEvent)
{
    UIXTree folderTree =
        feBean.getNavigatorManager().getFoldersNavigator().getFoldersTreeComponent();
    Object selectedPath =
        feBean.getNavigatorManager().getFoldersNavigator().getFirstSelectedTreePath();

    if (selectedPath != null)
    {
        TreeModel model =
            _feBean.getNavigatorManager().getFoldersNavigator().getFoldersTreeModel();
        Object oldRowKey = model.getRowKey();
        try
        {
            model.setRowKey(selectedPath);
            Object parentRowKey = model.getContainerRowKey();
            if (parentRowKey != null)
            {
                folderTree.getSelectedRowKeys().clear();
                folderTree.getSelectedRowKeys().add(parentRowKey);
                // This is an example of how to force a single attribute
                // to rerender. The method assumes that the client has an optimized
                // setter for "selectedRowKeys" of tree.
                FacesContext context = FacesContext.getCurrentInstance();
                ExtendedRenderKitService erks =
                    Service.getRenderKitService(context,
                        ExtendedRenderKitService.class);
                String clientRowKey = folderTree.getClientRowKeyManager().
                    getClientRowKey(context, folderTree, parentRowKey);
                String clientId = folderTree.getClientId(context);
                StringBuilder builder = new StringBuilder();
                builder.append("AdfPage.PAGE.findComponent('");
                builder.append(clientId);
                builder.append("').setSelectedRowKeys({'");
                builder.append(clientRowKey);
                builder.append("':true});");
                erks.addScript(context, builder.toString());
            }
        }
        finally
        {
```

```
        model.setRowKey(oldRowKey);
    }
    // Only really needed if using server-side rerendering
    // of the tree selection, but performing it here saves
    // a roundtrip (just one, to fetch the table data, instead
    // of one to process the selection event only after which
    // the table data gets fetched!)
    _feBean.getNavigatorManager().getFoldersNavigator().openSelectedFolder();
}
}
```

Using ADF Faces Client Behavior Tags

Client behavior tags in ADF Faces execute on the client side. ADF Faces provides a list of client behavior tags that you can use in place of client listeners.

ADF Faces client behavior tags provide declarative solutions to common client operations that you would otherwise have to write yourself using JavaScript, and register on components as client listeners. By using these tags instead of writing your own JavaScript code to implement the same operations, you reduce the amount of JavaScript code that needs to be downloaded to the browser.

ADF Faces provides these client behavior tags that you can use in place of client listeners:

- `panelDashboardBehavior`: Enables the runtime insertion of a child component into a `panelDashboard` component to appear more responsive. See [How to Use the panelDashboard Component](#).
- `insertTextBehavior`: Enables a command component to insert text at the cursor in an `inputText` component. See [How to Add the Ability to Insert Text into an inputText Component](#).
- `richTextEditorInsertBehavior`: Enables a command component to insert text (including preformatted text) at the cursor in a `richTextEditor` component. See [How to Add the Ability to Insert Text into a richTextEditor Component](#).
- `autoSuggestBehavior`: Enables list of values components to show items in a dropdown list that match what the user is typing. See [About List-of-Values Components](#).
- `showPopupBehavior`: Enables a command component to launch a popup component. See [Declaratively Invoking a Popup](#).
- `showPrintablePageBehavior`: Enables a command component to generate and display a printable version of the page. See [Displaying a Page for Print](#).
- `checkUncommittedDataBehavior`: Enables a command component to display a warning when the `immediate` attribute is set to true and a user attempts to navigate away from the page. For details see [Working with Navigation Components](#).
- `scrollComponentIntoViewBehavior`: Enables a command component to jump to a named component when clicked. See [How to Use the scrollComponentIntoViewBehavior Tag](#).

Tip:

ADF Faces also provides a server-side `scrollComponentIntoView` API that can be used when the component that is to be scrolled to may not yet be rendered on the page.

For example, if you have a table and you want to be able to scroll to a specific row, that row may be out of view when the table is first rendered. You can use the `scrollComponentIntoView` API as part of the data fetch event. See the *Java API Reference for Oracle ADF Faces*.

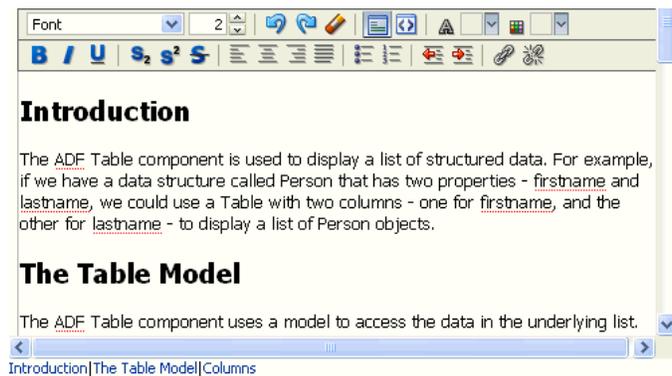
- `target`: Enables a component to declaratively execute or render a list of components when a specified event occurs. See [Using the Target Tag to Execute PPR](#).

Client behavior tags cancel server-side event delivery automatically. Therefore, any `actionListener` or `action` attributes on the parent component will be ignored. This cannot be disabled. If you want to also trigger server-side functionality, you should use either a client-side event (see [Using JavaScript for ADF Faces Client Events](#)), or add an additional client listener that uses `AdfCustomEvent` and `af:serverListener` to deliver a server-side event (see [Sending Custom Events from the Client to the Server](#)).

How to Use the `scrollComponentIntoViewBehavior` Tag

Use the `scrollComponentIntoViewBehavior` tag when you want the user to be able to jump to a particular component on a page. This action is similar to an anchor in HTML. For example, you may want to allow users to jump to a particular part of a page using a `commandLink` component. For the `richTextEditor` and `inlineFrame` components, you can jump to a subcomponent. For example, [Figure 6-1](#) shows a `richTextEditor` component with a number of sections in its text. The command links below the editor allow the user to jump to specific parts of the text.

Figure 6-1 `scrollComponentIntoViewBehavior` Tag in an Editor



You can also configure the tag to have focus switched to the component to which the user has scrolled.

Before you begin:

It may be helpful to have an understanding of behavior tags. See [Using ADF Faces Client Behavior Tags](#).

To use the `scrollComponentIntoViewBehavior` tag:

1. Create a command component that the user will click to jump to the named component. For procedures, see [How to Use Buttons and Links for Navigation and Deliver ActionEvents](#).
2. In the Components window, from the Operations panel, drag and drop a **Scroll Component Into View Behavior** as a child to the command component.
3. In the Insert Scroll Component Into View Behavior dialog, use the dropdown arrow to select **Edit** and then navigate to select the component to which the user should jump.
4. In the Properties window, set the `focus` attribute to `true` if you want the component to have focus after the jump.
5. For a `richTextEditor` or `inlineFrame` component, optionally enter a value for the `subTargetId` attribute. This ID is defined in the value of the `richTextEditor` or `inlineFrame` component.

For example, the value of the `subTargetId` attribute for the `scrollComponentIntoViewBehavior` tag shown in [Figure 6-1](#) is `Introduction`. The value of the `richTextEditor` is bound to the property shown in the following example. Note that `Introduction` is the ID for the first header.

```
private static final String _RICH_SECTIONED_VALUE =
    "<div>\n" +
    "    <h2>\n" +
    "        <a id=\"Introduction\"></a>Introduction</h2>\n" +
    "    <p>\n" +
    "        The ADF Table component is used to display a list of structured data.
    For example,\n" +
    "        if we have a data structure called Person that has two properties -
    firstname and\n" +
    "        lastname, we could use a Table with two columns - one for firstname,
    and the other\n" +
    "        for lastname - to display a list of Person objects.\n" +
    "    </p>\n" +
    " </div>\n" +
    " <div>\n" +
    "    <h2>\n" +
    "        <a id=\"The_Table_Model\"></a>The Table Model</h2>\n" +
    "    <p>\n" +
    "        . . .
    </div>";
```

Using Polling Events to Update Pages

The ADF Faces `poll` component can be used to deliver poll events to the server as a means to periodically update page components. To use the `poll` component, you must create a handler method to handle the functionality for the polling event.

ADF Faces provides the `poll` component whose `pollEvent` can be used to communicate with the server at specified intervals. For example, you might use the `poll` component to update an `outputText` component, or to deliver a heartbeat to the server to prevent users from being timed out of their session.

You need to create a listener for the `pollEvent` that will be used to do the processing required at poll time. For example, if you want to use the poll component to update the value of an `outputText` component, you would implement a `pollEventListener` method that would check the value in the data source and then update the component.

You can configure the interval time to determine how often the poll component will deliver its poll event. You also configure the amount of time after which the page will be allowed to time out. This can be useful, as the polling on a page causes the session to never time out. Each time a request is sent to the server, a session time out value is written to the page to determine when to cause a session time out. Because the poll component will continually send a request to the server (based on the interval time), the session will never time out. This is expensive both in network usage and in memory.

To avoid this issue, the `web.xml` configuration file contains the `oracle.adf.view.rich.poll.TIMEOUT` context-parameter, which specifies how long a page should run before it times out. A page is considered eligible to time out if there is no keyboard or mouse activity. The default timeout period is set at ten minutes. So if user is inactive for 10 minutes, that is, does not use the keyboard or mouse, then the framework stops polling, and from that point on, the page participates in the standard server-side session timeout (see [Session Timeout Warning](#)).

If the application does time out, when the user moves the mouse or uses the keyboard again, a new session timeout value is written to the page, and polling starts again.

You can override this time for a specific page using the poll component's `timeout` attribute.

How to Use the Poll Component

When you use the poll component, you normally also create a handler method to handle the functionality for the polling event.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Polling Events to Update Pages](#).

To use a poll component:

1. In a managed bean, create a handler for the poll event. For information about managed beans, see [Creating and Using Managed Beans](#).
2. In the Components window, from the Operations panel, drag and drop a **Poll** onto the page.
3. In the Properties window, expand the Common section and set the following:
 - **Interval:** Enter the amount of time in milliseconds between poll events. Set to 0 to disable polling.
 - **PollListener:** Enter an EL expression that evaluates to the method in Step 1.
 - **Timeout:** If you want to override the global timeout value in the `web.xml` file, set **Timeout** to the amount of time in milliseconds after which the page will stop polling and the session will time out.

7

Validating and Converting Input

This chapter describes how to add conversion and validation capabilities to ADF Faces input components in your application. It also describes how to add custom JSF conversion and validation, how to handle and display any errors, including those not caused by validation.

This chapter includes the following sections:

- [About ADF Faces Converters and Validators](#)
- [Conversion, Validation, and the JSF Lifecycle](#)
- [Adding Conversion](#)
- [Creating Custom ADF Faces Converters](#)
- [Adding Validation](#)
- [Creating Custom JSF Validation](#)

About ADF Faces Converters and Validators

ADF Faces provides the ability to both convert and validate user provided data to help ensure data integrity. Use the Converters to convert the values on ADF forms to the type that the application accepts them and use Validators to impose validations on the input components.

ADF Faces input components support conversion capabilities. A web application can store data of many types, such as `int`, `long`, and `date` in the model layer. When viewed in a client browser, however, the user interface has to present the data in a manner that can be read or modified by the user. For example, a date field in a form might represent a `java.util.Date` object as a text string in the format `mm/dd/yyyy`. When a user edits a date field and submits the form, the string must be converted back to the type that is required by the application. Then the data is validated against any rules and conditions. Conversely, data stored as something other than a `String` type can be converted to a `String` for display and updating. Many components, such as `af:inputDate`, automatically provide a conversion capability.

ADF Faces input components also support validation capabilities. You can add one or more validator tags to the component. In addition, you can create your own custom validators to suit your business needs.

Validators and converters have a default hint message that is displayed to users when they click in the associated field. For converters, the hint usually tells the user the correct format to use for input values, based on the given pattern. For validators, the hint is used to convey what values are valid, based on the validation configured for the component. If conversion or validation fails, associated error messages are displayed to the user. These messages can be displayed in dialogs, or they can be displayed on the page itself next to the component whose conversion or validation failed. For information about displaying messages in an ADF Faces application, see [Displaying Tips, Messages, and Help](#).

ADF Faces converters is a set of converters that extends the standard JSF converters. Since ADF Faces converters for input components operate on the client-side, errors in conversion can be caught at the client and thus avoid a round trip to the server. You can easily drag and drop ADF Faces converters into an input component.

ADF Faces validators also augment the standard JSF validators. ADF Faces validators can operate on both the client and server side. The client-side validators are in written JavaScript and validation errors caught on the client-side can be processed without a round-trip to the server.

ADF Faces Converters and Validators Use Cases and Examples

You use ADF Faces converters to convert input from an input component into the format the model expects. A typical use case is using an input component for entering numbers and including a converter to convert the string entered by the user into a number for the model to process. For example, an `af:inputText` component is used for a product Id attribute. You add the `af:convertNumber` converter to the `af:inputText` component to convert from `String` to `Number`. Another example is when you have an `inputText` component for an attribute for the cost of a product. You can use `af:convertNumber` to convert the input string into the proper currency format.

You add validators to input components in the same way to validate the input string. For instance, you can add a validator to the `af:inputText` component to check that the number of digits for the product Id are within the proper range. You add `af:validateLength` to `af:inputText` and set the `minimum` and `maximum` attributes to define the valid digit length.

Additional Functionality for ADF Faces Converters and Validators

You may find it helpful to understand other ADF Faces features before you implement your converters and validators. Following are links to other sections that may be useful.

- For detailed information about how conversion and validation works in the JSF Lifecycle, see [Using the JSF Lifecycle with ADF Faces](#) .
- ADF Faces lets you customize the detail portion of a conversion error message instead of a default message. For information about creating messages, see [Displaying Tips, Messages, and Help](#).
- Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).

Conversion, Validation, and the JSF Lifecycle

The conversion and validation capabilities supported by ADF Faces are basically performed at different phases of JSF lifecycle. You can learn about the phase at which conversion and validation takes place in JSF lifecycle and also learn about what happens when the conversion or validation fails.

When a form with data is submitted, the browser sends a request value to the server for each UI component whose `editable value` attribute is bound. Request values are decoded during the JSF Apply Request Values phase and the decoded value is saved locally on the component in the `submittedValue` attribute. If the value requires

conversion (for example, if it is displayed as a `String` type but stored as a `java.util.Date` object), the data is converted to the correct type during the Process Validation phase on a per-UI-component basis.

If validation or conversion fails, the lifecycle proceeds to the Render Response phase and a corresponding error message is displayed on the page. If conversion and validation are successful, then the Update Model phase starts and the converted and validated values are used to update the model.

When a validation or conversion error occurs, the component whose validation or conversion failed places an associated error message in the queue and invalidates itself. The current page is then redisplayed with an error message. ADF Faces components provide a way of declaratively setting these messages.

For detailed information about how conversion and validation works in the JSF Lifecycle, see [Using the JSF Lifecycle with ADF Faces](#) .

Adding Conversion

ADF Faces converters are used to convert input from an input component into the format the model expects. You can use JDeveloper to automatically insert a converter to the UI component or you can manually insert a converter into a UI component.

A web application can store data of many types (such as `int`, `long`, `date`) in the model layer. When viewed in a client browser, however, the user interface has to present the data in a manner that can be read or modified by the user. For example, a date field in a form might represent a `java.util.Date` object as a text string in the format `mm/dd/yyyy`. When a user edits a date field and submits the form, the string must be converted back to the type that is required by the application. You can set only one converter on a UI component.

When you create an `af:inputText` component and set an attribute that is of a type for which there is a converter, JDeveloper automatically adds that converter's tag as a child of the input component. This tag invokes the converter, which will convert the `String` type entered by the user back into the type expected by the object.

The JSF standard converters, which handle conversion between `String` types and simple data types, implement the `javax.faces.convert.Converter` interface. The supplied JSF standard converter classes are:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `EnumConverter`
- `FloatConverter`
- `IntegerConverter`

- LongConverter
- NumberConverter
- ShortConverter

Table 7-1 shows the converters provided by ADF Faces.

Table 7-1 ADF Faces Converters

Converter	Tag Name	Description
ColorConverter	af:convertColor	Converts <code>java.lang.String</code> objects to <code>java.awt.Color</code> objects. You specify a set of color patterns as an attribute of the converter.
DateTimeConverter	af:convertDateTime	Converts <code>java.lang.String</code> objects to <code>java.util.Date</code> objects. You specify the pattern and style of the date as attributes of the converter.
NumberConverter	af:convertNumber	Converts <code>java.lang.String</code> objects to <code>java.lang.Number</code> objects. You specify the pattern and type of the number as attributes of the converter.

As with validators, the ADF Faces converters are also run on the client side.

If no converter is explicitly added, ADF Faces will attempt to create a converter based on the data type. Therefore, if the value is bound to any of the following types, you do not need to explicitly add a converter:

- `java.util.Date`
- `java.util.Color`
- `java.awt.Color`
- `java.lang.Number`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.Byte`
- `java.lang.Float`
- `java.lang.Double`

Unlike the converters listed in Table 7-1, the JavaScript-enabled converters are applied by `type` and used instead of the standard ones, overriding the `class` and `id` attributes. They do not have associated tags that can be nested in the component.

How to Add a Converter

You can also manually insert a converter into a UI component.

Before you begin:

It may be helpful to have an understanding of converters. See [Adding Conversion](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for ADF Faces Converters and Validators](#).

To add ADF Faces converters that have tags:

1. In the Structure window, right-click the component for which you would like to add a converter, choose **Insert Inside** *component*, then **ADF Faces** to insert an ADF Faces converter.

You may also choose **JSF > Converter** to insert a JSF converter.

2. Choose a converter tag (for example, **Convert Date Time**) and click **OK**.
3. In the JSF page, select the component, and in the Properties window set values for the attributes, including any messages for conversion errors. For additional help, right-click any of the attributes and choose **Help**.

You can set multiple patterns for some ADF Faces converters. See [How to Specify Multiple Converter Patterns](#).

ADF Faces lets you customize the detail portion of a conversion error message. By setting a value for a **MessageDetailxyz** attribute, where **xyz** is the conversion error type (for example, `MessageDetailconvertDate`), ADF Faces displays the custom message instead of a default message, if conversion fails. For information about creating messages, see [Displaying Tips, Messages, and Help](#).

How to Specify Multiple Converter Patterns

Some converters support multiple patterns. Patterns specify the format of data accepted for conversion. Multiple patterns allow for more than one format. For example, a user could enter dates using a slash (/) or hyphen (-) as a separator. Note that not all converters support multiple patterns, although pattern matching is flexible and multiple patterns may not be needed.

The following example illustrates the use of a multiple pattern for the `af:convertColor` tag in which "255-255-000" and "FFFF00" are both acceptable values.

```
<af:inputColor colorData="#{adfFacesContext.colorPalette.default49}" id="sic3"
  label="Select a color" value="#{demoColor.colorValue4}" chooseId="chooseId">
  <af:convertColor patterns="rrr-ggg-bbb RRGGBB #RRGGBB"
    transparentAllowed="false"/>
</af:inputColor>
```

The following example illustrates the use of an `af:convertDateTime` tag in which "6/9/2007" and "2007/9/6" are both acceptable values.

```
<af:inputDate id="mdf5" value="2004/09/06" label="attached converter">
  <af:convertDateTime pattern="yyyy/M/d" secondaryPattern="d/M/yyyy" />
</af:inputDate>
```

The following example illustrates an `af:convertNumber` tag with the `type` attribute set to `currency` to accept "\$78.57" and "\$078.57" as values for conversion.

```
<af:inputText label="type=currency" value="#{validate.currency}" id="it1">
  <af:convertNumber type="currency"/>
</af:inputText>
```

How to Specify Negative Numbers for Converters

Negative numbers are used in financial applications and have attribute level support for client-side conversion using the `af:convertNumber` tag with the **negativePrefix** and **negativeSuffix** attributes to format negative values using specified prefix and suffix characters. For example, financial applications often format negative numbers using parentheses, where (99.00) represents the negative value for 99.00.

These attributes accomplish the same thing as specifying a converter pattern on the number converter, like `#;(#)`, where the characters following the ";" in the pattern indicate the negative pattern and the open and close parenthesis in the expression represent the negative prefix and suffix respectively for negative numbers. However, converter patterns are not processed on the client side and require a server roundtrip to format the output.

You can eliminate the need for a server roundtrip to perform the conversion using the `af:convertNumber` tag. This client-side conversion will override any implied negative prefix/suffix using the converter pattern.

The following example illustrates an `af:convertNumber` tag with the **negativePrefix** and **negativeSuffix** attributes set to "(" and ")" respectively to accept values a user might enter such as "-99.00" and convert them to "(99.00)". With this setting, the conversion is handled on the client as soon as the user tabs out of the text input.

```
<af:inputText label="type=number" id="it3">
  <af:convertNumber negativePrefix="(" negativeSuffix=")" type="number"/>
</af:inputText>
```

What Happens at Runtime: How Converters Work

When the user submits the page containing converters, the ADF Faces `validate()` method calls the converter's `getAsObject()` method to convert the `String` value to the required object type. When there is not an attached converter and if the component is bound to a bean property in the model, then ADF checks the model's data type and attempts to find the appropriate converter. If conversion fails, the component's `valid` attribute is set to `false` and JSF adds an error message to a queue that is maintained by `FacesContext`. If conversion is successful and there are validators attached to the component, the converted value is passed to the validators. If no validators are attached to the component, the converted value is stored as a local value that is later used to update the model.

What You May Need to Know About Number Converters

`af:convertNumber` supports either server-side or client-side conversion. The converter first attempts to perform number formatting, parsing and conversions on the fly within the browser thereby eliminating the need to perform a server roundtrip for formatting input numbers. However, client conversion is skipped in the following situations:

- If the `pattern` attribute is specified. The converter will only run on the server.
- If the input string contains more than 15 digits, which is the maximum precision supported by JavaScript number. The converter will only run on the server.

When `af:convertNumber` displays an input value with more decimal digits than specified by `maxFractionDigits`, by default it uses Java's `HALF_EVEN` method to round off the value.

If you want `af:convertNumber` to use a method other than `HALF_EVEN` (such as `HALF_UP` or `FLOOR`), follow these tips:

- To configure a particular instance of `af:convertNumber`, set the **roundingMode** attribute to the desired value.

For example:

```
<af:convertNumber roundingMode="FLOOR" ... />
```

- To configure all instances of `af:convertNumber` in your application, add the **rounding-mode** attribute in `trinidad-config.xml` and set it accordingly.

For example:

```
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
...
  <rounding-mode>FLOOR</rounding-mode>
</trinidad-config>
```

The value of **roundingMode** must be supported by Java's `RoundingMode`. The value can be specified as a string in the `.jspx` or the config file, or as an EL expression bound to a method that returns `RoundingMode` type value.

Note:

Input value cannot be rounded at client-side unless the `roundingMode` is set to `half-up`. If `roundingMode` is not set to `half-up`, the input number is retained unaltered at client-side for a subsequent postback to perform the actual rounding on the server-side.

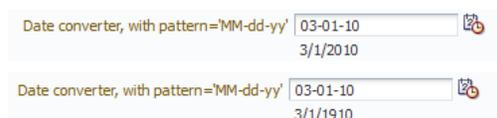
For more information about `RoundingMode`, see the Java API documentation.

What You May Need to Know About Date Time Converters

You should use a four-digit year pattern with a date converter to avoid ambiguity. If you are using a two-digit year format as the pattern, all four-digit year values appear as two digit year values. For example, if you are using a two-digit year format (such as `MM-dd-yy`) as the pattern, the date values `03-01-1910` and `03-01-2010` appear as `03-01-10` in the input field and could be interpreted incorrectly, though the server stores the correct year value.

Figure 7-1 shows the date values as they appear in the `inputDate` component, with an `outputText` component below that shows the original values stored on the server.

Figure 7-1 Date Converter With Two-Digit Year Format



If you are using a two-digit year format, all strings containing two-digit years will be resolved into a date within `two-digit-year-start` and `two-digit-year-start + 100`. For example, if `two-digit-year-start` value is 1912, the string 01/01/50 gets resolved to 01/01/1950. To enter dates outside this range, the end user should enter a date with the full (four-digit) year. For information about `two-digit-year-start` element and how to configure it, see [What You May Need to Know About Elements in trinidad-config.xml](#).

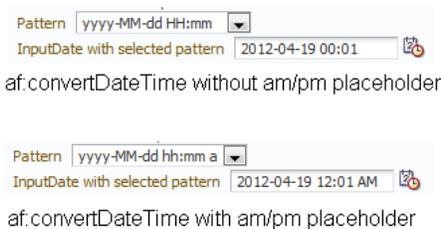
 **Note:**

While using a two-digit year format, two digit years will be placed in the range determined by `two-digit-year-start` even if the user is editing an existing value.

For example, assuming `two-digit-year-start` is set to 1950 (resolving year values to the range 1950 through 2050) and the `inputDate` component has value 03/01/1776 (displayed as 03/01/76). If the user modifies the value to 03/01/77, the new date would be 03/01/1977, not 03/01/1777 as may be expected.

If you want to use a 12-hour format (for example, `MM/dd/yyyy - hh:mm`) as the pattern with `af:convertDateTime`, you should also include the `am/pm` placeholder in the pattern (for example, `MM/dd/yyyy - hh:mm a`), otherwise the picker will not show `am/pm` options and the user will not be able to save the `am/pm` information. [Figure 7-2](#) shows the `inputDate` component with and without `am/pm` placeholders in their patterns.

Figure 7-2 Using AM/PM Placeholder With `af:convertDateTime`



If you want to display timezone information by including the `timezone` placeholder (`z`) in the `convertDateTime` pattern (example, `yyyy-MM-dd hh:mm:ss a z`), note that the `trinidad convertDateTime` overrides the display of the timezone names. When used in conjunction with `inputText` or `outputText`, the converter displays the timezone in the format `GMT + x`. When used in conjunction with `inputDate`, the timezone is displayed from a selection of pre-configured display values, such as `(UTC-08:00) Los Angeles - Pacific Time (PT)`.

Creating Custom ADF Faces Converters

You can create your own converters to meet your specific business needs. There are different procedures to create custom ADF Faces server-side converters or client-side converters.

Custom JSF converters run on the server-side using Java. ADF Faces Converters behave as JSF converters, but also support client-side conversion and validation using Javascript.

How to Create a Custom ADF Faces Converter

Creating a custom ADF Faces converter requires writing the business logic for the conversion and then registering the custom converter with the application. To use the custom ADF Faces converter, you use the `f:converter` tag and set the custom ADF Faces converter as a property of that tag, or you can use the `converter` attribute on the input component to bind to that converter.

To create a server-side converter, you must:

- [Implement Server-Side \(Java\) Conversion](#)
- [Register ADF Faces Converter in faces-config.xml](#)

The ADF Framework supports client-side conversion and validation to minimize postbacks to the server. Client-side implementation is generally optional - if no client-side implementation is available, the framework will simply invoke the server-side converter during postback, just as in JSF.

Client-side implementation is required if the converter is used in conjunction with certain components which support client interaction without postback, such as `inputNumberSlider`, `inputNumberSpinbox`, and `inputDate`. To determine if a client converter is required for a component, refer to the component's tagdoc.

ADF Faces client-side converters work in the same way standard JSF conversion works on the server, except that JavaScript is used on the client. JavaScript converter objects can throw `ConverterException` exceptions and they support the `getAsObject()` and `getAsString()` methods.

To create a client-side converter, you must:

- [Create a Client-Side Version of the Converter](#)
- [Modify the Server Converter to Enable Client Conversion](#)

Implement Server-Side (Java) Conversion

ADF Faces converters are implemented on the server-side similarly to custom JSF converters.

1. Create a Java class that implements the `javax.faces.converter.Converter` interface. The implementation must contain a public no-args constructor, a set of accessor methods for any attributes, and `getAsObject` and `getAsString` methods to implement the `Converter` interface.

The `getAsObject()` method takes the `FacesContext` instance, the UI component, and the `String` value to be converted to a specified object, for example:

```
public Object getAsObject(FacesContext context,
                        UIComponent component,
                        java.lang.String value){
    ..
}
```

The `getAsString()` method takes the `FacesContext` instance, the UI component, and the object to be converted to a `String` value, for example:

```
public String getAsString(FacesContext context,
                        UIComponent component,
                        Object value){
    ..
}
```

For information about these classes, refer to the Javadoc or visit <http://docs.oracle.com/javaee/index.html>.

2. Add the needed conversion logic. This logic should use `javax.faces.convert.ConverterException` to throw the appropriate exceptions and `javax.faces.application.FacesMessage` to generate the corresponding error messages.

For information about the `Converter` interface and the `FacesMessage` error handlers, see the Javadoc for `javax.faces.convert.ConverterException` and `javax.faces.application.FacesMessage`, or visit <http://docs.oracle.com/javaee/index.html>.

If your application saves state on the client, the custom ADF Faces converter must implement the `Serializable` interface or the `StateHolder` interface, and the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods of the `StateHolder` interface. See the Javadoc for the `StateHolder` interface of `javax.faces.component` package, or visit <http://docs.oracle.com/javaee/index.html>.

Register ADF Faces Converter in faces-config.xml

You should register the converter in the `faces-config.xml` file.

1. Open the `faces-config.xml` file.
The `faces-config.xml` file is located in the **View_Project > WEB-INF** node in the JDeveloper Applications window.
 2. In the editor window, click the **Overview** tab.
 3. Choose **Converters** and click **Add** to enter the converter information.
- Click **Help** or press F1 for additional help in registering the converter.

Create a Client-Side Version of the Converter

Write a client JavaScript version of the converter, passing relevant information to a constructor, as shown in the following example.

```
function TrConverter()
{
}

/**
```

```

* Convert the specified model object value, into a String for display
* @param value Model object value to be converted
* @param label label to identify the editableValueHolder to the user
* @return the value as a string or undefined in case of no converter mechanism is
* available (see TrNumberConverter).
*/
TrConverter.prototype.getAsString = function(value, label){}

/**
* Convert the specified string value into a model data object
* which can be passed to validators
* @param value String value to be converted
* @param label label to identify the editableValueHolder to the user
* @return the converted value or undefined in case of no converter mechanism is
* available (see TrNumberConverter).
*/

TrConverter.prototype.getAsObject = function(value, label){}

```

If errors are encountered, the client can throw a `TrConverterException` exception to show a `TrFacesMessage` error message.

The following example shows the signature for `TrFacesMessage`.

```

/**
* Message similar to javax.faces.application.FacesMessage
* @param summary - Localized summary message text
* @param detail - Localized detail message text
* @param severity - An optional severity for this message. Use constants
* SEVERITY_INFO, SEVERITY_WARN, SEVERITY_ERROR, and
* SEVERITY_FATAL from the FacesMessage class. Default is
* SEVERITY_INFO
*/
function TrFacesMessage(summary,detail,severity){
  ..
}

```

The following example shows the signature for `TrFacesException`.

```

/**
* TrConverterException is an exception thrown by the getAsObject() or getAsString()
* method of a Converter, to indicate that the requested conversion cannot be
* performed.
* @param facesMessage the TrFacesMessage associated with this exception
* @param summary Localized summary message text, used to create only if facesMessage
* is null
* @param detail Localized detail message text, used only if facesMessage is null
*/
function TrConverterException(facesMessage, summary, detail){
  ..
}

```

Modify the Server Converter to Enable Client Conversion

Change the server converter to implement `ClientConverter` interface, which indicates that the class supports client-side conversion. For information on the interface methods, see the `ClientConverter` javadoc on <http://myfaces.apache.org>.

The following example shows a custom javascript converter implementation for social security number.

```
function ssnGetAsString(value)
{
    return value.substring(0,3) + '-' +
        value.substring(3,5) + '-' +
        value.substring(5);
}

function ssnGetAsObject(value)
{
    if (!value)
        return (void 0);

    var len=value.length;
    var messageKey = SSNConverter.NOT;
    if (len < 9 )
        messageKey = SSNConverter.SHORT;
    else if (len > 11)
        messageKey = SSNConverter.LONG;
    else if (len == 9)
    {
        if (!isNaN(value))
            return value;
    }
    else if (len == 11 && value.charAt(3) == '-' && value.charAt(6) == '-')
    {
        var result = value.substring(0,3) +
            value.substring(4,6) +
            value.substring(7);

        if (!isNaN(result))
            return result;
    }

    if (messageKey!=void(0) && this._messages!=void(0))
        return new ConverterException(this._messages[messageKey]);

    return (void 0);
}

function SSNConverter(messages)
{
    this._messages = messages;
}

SSNConverter.prototype = new Converter();
SSNConverter.prototype.getAsString = ssnGetAsString;
SSNConverter.prototype.getAsObject = ssnGetAsObject;

SSNConverter.SHORT = 'S';
SSNConverter.LONG  = 'L';
SSNConverter.NOT   = 'N';
```

The following example shows a custom Java class that implements server implementation for social security number. The details of the Java code has been removed from the `getAsObject()` and `getAsString()` methods.

```
package oracle.adfdemo.view.faces.convertValidate;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
```

```
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

import oracle.adf.view.faces.converter.ClientConverter;
/**
 * <p>Social Security number converter.</p>
 *
 */
public class SSNConverter implements Converter, ClientConverter
{
    public static final String CONVERTER_ID = "oracle.adfdemo.SSN";

    public Object getAsObject(
        FacesContext context,
        UIComponent component,
        String value)
    {
        // some Java code ...
    }

    public String getAsString(
        FacesContext context,
        UIComponent component,
        Object value)
    {
        // some Java code ...
    }

    public String getClientConversion(
        FacesContext context,
        UIComponent component)
    {
        // in a real app the messages would be translated
        return "new SSNConverter({" +
            "S:'Value \"{0}\" in \"{1}\" is too short.'," +
            "L:'Value \"{0}\" in \"{1}\" is too long.'," +
            "N:'Value \"{0}\" in \"{1}\" " +
            "is not a social security number.'})";
    }

    public String getClientScript(
        FacesContext context,
        UIComponent component)
    {
        // check if the script has already been returned this request
        Object scriptReturned =
            context.getExternalContext().getRequestMap().get(CONVERTER_ID);

        // if scriptReturned is null the script hasn't been returned yet
        if ( scriptReturned == null)
        {
            context.getExternalContext().getRequestMap().put(CONVERTER_ID,
                Boolean.TRUE);
            return _sSSNjs;
        }
        // if scriptReturned is not null, then script has already been returned,
        // so don't return it again.
        else
            return null;
    }
}
```

```

    }
    private static final String _sSSNjs =
"function ssnGetAsString(value)"+
"{return value.substring(0,3) + '-' " +
  "+ value.substring(3,5) + '-' + value.substring(5);}" +
"function ssnGetAsObject(value)" +
  "{if (!value)return (void 0);" +
  "var len=value.length;" +
  "var messageKey = SSNConverter.NOT;" +
  "if (len < 9)"+
    "messageKey = SSNConverter.SHORT;" +
  "else if (len > 11)"+
    "messageKey = SSNConverter.LONG;" +
  "else if (len == 9)" +
  "{ if (!isNaN(value))" +
    "return value;" +
  }" +
  "else if (len == 11 && value.charAt(3) == '-' && " +
  "value.charAt(6) == '-')" +
  "{" +
  "var result = value.substring(0,3) + value.substring(4,6) + " +
  "value.substring(7);"+
  "if (!isNaN(result))"+
    "return result;" +
  }" +
  "if (messageKey!=void(0) && this._messages!=void(0))" +
    "return new ConverterException(this._messages[messageKey]);" +
  "return void(0);}" +
"function SSNConverter(messages)" +
  "{this._messages = messages;}" +
"SSNConverter.prototype = new Converter();" +
"SSNConverter.prototype.getAsString = ssnGetAsString;" +
"SSNConverter.prototype.getAsObject = ssnGetAsObject;" +
"SSNConverter.SHORT = 'S';" +
"SSNConverter.LONG = 'L';" +
"SSNConverter.NOT = 'N';";
}

```

Using Custom ADF Faces Converter on a JSF Page

If a custom ADF Faces converter is registered in an application under a class for a specific data type, whenever a component's value references a value binding that has the same type as the custom converter object, JSF will automatically use the converter of that class to convert the data. In that case, you do not need to use the converter attribute to register the custom converter on a component, as shown in the following code:

```
<af:inputText value="#{myBean.myProperty}"/>
```

where `myProperty` data type has the same type as the custom converter. Alternatively, you can bind your converter class to the converter attribute of the input component.

What You May Need to Know About Custom ADF Faces Converters

When you define value formatting for a table or chart, the value formatting applies only on the non-active events and does not apply on the ADS event. The converters, which are responsible for formatting response data do not work on ADS response. To format

the ADS data, you can use the `oracle.adf.view.rich.ads.USE_COMPONENT_FORMATTER` parameter in the `web.xml` file. You can set the following values for this parameter:

- **NEVER:** Apply this option to never use the component converter.
- **IF_MODEL_DID_NOT_FORMAT:** Apply this option to use the component converter only if the new value and the formatted value from the active model are identical.
- **ALWAYS:** Apply this option to use the component converter if it implements `formatterFactory` and returns a not null formatted value.

How to Declaratively Register a Client-Side Only Converter Script

The `af:clientConverter` tag provides UI developers with a declarative alternative to register a custom client-only converter script, while still allowing the use of pre-programmed methods.

Usually, input or output fields whose data is encrypted by cloud data protection services cannot be formatted on the server for the risk of data corruption. Such fields are generally identified by the `protected` attribute. The `af:clientConverter` tag is useful in situations where the data formatting requirements needs be delayed to the point of page rendering where it is safe to format protected data.

The tag is applied as a child to input or output tags where conversion is needed. It uses the `getAsString` and `getAsObject` attributes to accept an inline JavaScript expression. The tag also offers the `getAsStringMethod` and `getAsObjectMethod` attributes to call a JavaScript method instead.

To use the `af:clientConverter` tag:

1. Set `clientComponent` to `true` on the parent input or output tag. Presence of `clientComponent` is mandatory for the functioning of the client converter.
2. Add an `af:clientConverter` tag to the input or output element to be formatted.
3. To convert an input to a desired string format, provide the valid JavaScript expression in the `getAsString` attribute. Use the `value` variable to access the field value.

The following example accepts a numeric input and formats it as a credit card number.

```
<af:outputText value="1234567890123456" id="outputText1"
clientComponent="true">
<af:clientConverter getAsString="return value.substr(0,4) + '-' +
value.substr(4,4) + '-' + value.substr(8,4) + '-' +
value.substr(12,4);" />
</af:outputText>
```

4. To convert a formatted string back to its original form, provide the valid JavaScript expression in the `getAsObject` attribute.

This is useful for internal validation purposes. For example, the credit card field should have a length restriction validator for a maximum allowed length of 16. In the absence of a `getAsObject` implementation, the validator would receive "1234-

5678–9012–3456” which has a length of 19 and would fail the validation. The following example accepts a formatted input and returns a numeric value.

```
<af:inputText value="1234567890123456" id="inputText1"
clientComponent="true">
<af:clientConverter getAsString="return value.substr(0,4) + '-' +
value.substr(4,4) + '-' + value.substr(8,4) + '-' + value.substr(12,4);"
getAsObject="return (value == null || value.length == 0)? value :
value.replace(/-/g, '');"/>
</af:inputText>
```

 **Note:**

If the converter is applied to an input field, the submitted value is taken from the field and not from the return value of `getAsObject/getAsObjectMethod`.

5. As an alternative to using JavaScript inline, you can use previously created methods. To do this, use the `getAsStringMethod` and `getAsObjectMethod` attributes.

If the expression language is too verbose, or if a method exists for reuse, these two attributes may be used. In the following example, the method `formatCreditCardNumber()` is called to tokenize a string input as a credit card number. The method `parseCreditCardNumber()` may be invoked to return the tokenized number to its original format.

 **Note:**

The implementation of the `getAsObject/getAsObjectMethod` attributes is optional if the parent tag has no associated validator.

```
<af:inputText value="#{demoInput.creditCardNumber}" id="inpt1">
<af:clientConverter getAsStringMethod="formatCreditCardNumber"
getAsObjectMethod="parseCreditCardNumber" hint="Credit Card digits"/>
</af:inputText>
```

The following example shows the sample methods `formatCreditCardNumber()` and `parseCreditCardNumber()`. Methods can be used inline with the `af:resource` tag or be part of a linked external JavaScript file.

```
<af:resource type="javascript">
/**
 * Formats a given credit card number string into a standard 4 digit
space separated grouping. (xxx xxx xxx xxx)
 * @param {String} The unformatted credit card number string that
needs to be formatted
 *
 * @return {String} The formatted credit card number
 */
```

```

function formatCreditCardNumber(value)
{
    if (value == null || value.length == 0)
        return value;

    var retVal = value.substr(0,4) + ' ';
    retVal += value.substr(4,4) + ' ';
    retVal += value.substr(8,4) + ' ';
    retVal += value.substr(12,4);

    return retVal;
}

/**
 * Parses the credit card number as formatted by the function
 * formatCreditCardNumber back to a contiguous numeric string for postback.
 * @param {value} The formatted credit card number string that needs
 * to be parsed
 *
 * @return {String} The parsed credit card number
 */
function parseCreditCardNumber(value)
{
    if (value == null || value.length == 0)
        return value;

    return value.replace(/ /g, '');
}
</af:resource>

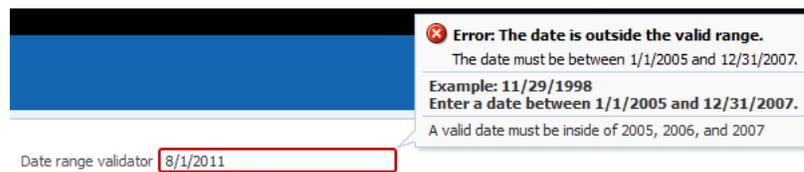
```

Adding Validation

ADF Faces input components support validation capabilities. You can add validation for an input component using UI component attributes, Default ADF Faces validators, and Custom ADF Faces validators.

You can add validation so that when a user edits or enters data in a field and submits the form, the data is validated against any set rules and conditions. If validation fails, the application displays an error message. For example, in [Figure 7-3](#) a specific date range for user input with a message hint is set by the `af:validateDateTimeRange` component and an error message is displayed in the message popup window when an invalid value is entered.

Figure 7-3 Date Range Validator with Error Message



On the view layer use ADF Faces validation when you want client-side validation. All validators provided by ADF Faces have a client-side peer. Many components have attributes that provide validation. See [Using Validation Attributes](#). In addition, ADF Faces provides separate validation classes that can be run on both the client and the server. For details, see [Using ADF Faces Validators](#). You can also create your own validators. For information about custom validators, see [How to Create a Custom JSF Validator](#).

How to Add Validation

By default, ADF Faces syntactic and semantic validation occurs on both the client and server side. Client-side validation allows validators to catch and display data without requiring a round-trip to the server.

ADF Faces provides the following types of validation:

- **UI component attributes:** ADF Faces input components provide attributes that can be used to validate data. For example, you can supply simple validation using the `required` attribute on ADF Faces input components to specify whether or not a value must be supplied. When the `required` attribute is set to `true`, the component must have a value. Otherwise the application displays an error message. See [Using Validation Attributes](#).
- **Default ADF Faces validators:** The validators supplied by the JSF framework provide common validation checks, such as validating date ranges and validating the length of entered data. See [Using ADF Faces Validators](#).
- **Custom ADF Faces validators:** You can create your own validators and then select them to be used in conjunction with UI components. See [Creating Custom JSF Validation](#).

When validation is added, validation errors can be displayed inline or in a popup window on the page. For information about displaying messages created by validation errors, see [Displaying Tips, Messages, and Help](#).

Using Validation Attributes

Many ADF Faces UI components have attributes that provide simple validation. For example, the `af:inputDate` component has `maxValue` and `minValue` attributes to specify the maximum and minimum number allowed for the Date value.

For additional help with UI component attributes, in the Properties window, right-click the attribute name and choose **Help**.

Using ADF Faces Validators

ADF Faces Validators are separate classes that can be run on the server or client. [Table 7-2](#) describes the validators and their logic.

Table 7-2 ADF Faces Validators

Validator	Tag Name	Description
ByteLengthValidator	af:validateByteLength	Validates the byte length of strings when encoded. The <code>maxLength</code> attribute of <code>inputText</code> is similar, but it limits the number of characters that the user can enter.
DateRestrictionValidator	af:validateDateRestriction	Validates that the entered date is valid with some given restrictions.
DateTimeRangeValidator	af:validateDateTimeRange	Validates that the entered date is within a given range. You specify the range as attributes of the validator.
DoubleRangeValidator	af:validateDoubleRange	Validates that a component value is within a specified range. The value must be convertible to a floating-point type.
LengthValidator	af:validateLength	Validates that the length of a component value is within a specified range. The value must be of type <code>java.lang.String</code> .
LongRangeValidator	af:validateLongRange	Validates that a component value is within a specified range. The value must be any numeric type or <code>String</code> that can be converted to a long data type.
RegExpValidator	af:validateRegExp	Validates the data using Java regular expression syntax.

 **Note:**

To register a custom validator on a component, use a standard JSF `f:validator` tag. For information about using custom validators, see [Creating Custom JSF Validation](#).

To add ADF Faces validators:

1. In the Structure window, right-click the component for which you would like to add a validator, choose **Insert Inside component**, then **ADF Faces** to insert an ADF Faces validator.

You may also choose **JSF > Validator** to insert a JSF reference implementation validator.

2. Choose a validator tag (for example, **Validate Date Time Range**) and click **OK**.
3. In the JSF page, select the component and in the Properties window, set values for the attributes, including any messages for validation errors that may occur, such as empty or incorrect required values.

For additional help, right-click any of the attributes and choose **Help**.

ADF Faces lets you customize the detail portion of a validation error message. By setting a value for a **MessageDetailxyz** attribute, where **xyz** is the validation error type (for example, `MessageDetailmaximum`), ADF Faces displays the custom message instead of a default message, if validation fails.

What Happens at Runtime: How Validators Work

When the user submits the page, ADF Faces checks the submitted value and runs conversion on any non-null value. The converted value is then passed to the `validate()` method. If the value is empty, the `required` attribute of the component is checked and an error message is generated if indicated. If the submitted value is non-null, the validation process continues and all validators on the component are called in order of their declaration.



Note:

ADF Faces provides extensions to the standard JSF validators, which have client-side support.

ADF Faces validation is performed during the Process Validations phase. If any errors are encountered, the components are invalidated and the associated messages are added to the queue in the `FacesContext` instance. The Update Model phase only happens when if there are no errors converting or validating. Once all validation is run on the components, control passes to the model layer, which runs the Validate Model Updates phase. As with the Process Validations phase, if any errors are encountered, the components are invalidated and the associated messages are added to the queue in the `FacesContext` instance.

The lifecycle then goes to the Render Response phase and redisplay the current page. If the component generates an error, ADF Faces automatically highlights the error. For instance, ADF Faces renders a red box around an `inputText` component when there is a validation error, as shown in [Figure 7-4](#).

Figure 7-4 Validation Error

The screenshot shows a JSF form with the following fields and values:

- DepartmentId: 10
- DepartmentName: bad (highlighted with a red border, indicating a validation error)
- ManagerId: 200 (dropdown menu)
- LocationId: 1700

Navigation buttons: First, Previous, Next, Last, and a Submit button.

For information about adding error messages when a validation or conversion error occurs, see [Displaying Hints and Error Messages for Validation and Conversion](#).

What You May Need to Know About Multiple Validators

You can set zero or more validators on a UI component. You can set the `required` attribute and use validators on a component. However, if you set the `required` attribute to `true` and the value is `null` or a zero-length string, the component is invalidated and any other validators registered on the component are not called.

This combination might be an issue if there is a valid case for the component to be empty. For example, if the page contains a **Cancel** button, the user should be able to click that button and navigate off the page without entering any data. To handle this case, you set the `immediate` attribute on the **Cancel** button's component to `true`. This attribute allows the action to be executed during the Apply Request Values phase. Then the default JSF action listener calls `FacesContext.renderResponse()`, thus bypassing the validation whenever the action is executed. See [Using the JSF Lifecycle with ADF Faces](#).

Creating Custom JSF Validation

In ADF Faces, you can add your own validation logic to meet your specific business needs. You can create custom validation by adding a method that provides the required validation on a backing bean or by using JSF validators.

If you want custom validation logic for a component on a single page, you can create a validation method on the page's backing bean.

If you want to create logic that will be reused by various pages within the application, or if you want the validation to be able to run on the client side, you should create a JSF validator class. You can then create an ADF Faces version, which will allow the validator to run on the client.

How to Create a Backing Bean Validation Method

When you want custom validation for a component on a single page, create a method that provides the required validation on a backing bean.

Before you begin:

It may be helpful to have an understanding of custom JSF validation. See [Creating Custom JSF Validation](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for ADF Faces Converters and Validators](#).

To add a backing bean validation method:

1. Insert the component that will require validation into the JSF page.
2. In the visual editor, double-click the component.
3. In the Bind Validator Property dialog, enter or select the managed bean that will hold the validation method, or click **New** to create a new managed bean. Use the default method signature provided or select an existing method if the logic already exists.

When you click **OK** in the dialog, JDeveloper adds a skeleton method to the code and opens the bean in the source editor.

4. Add the required validation logic. This logic should use the `javax.faces.validator.ValidatorException` exception to throw the appropriate exceptions and the `javax.faces.application.FacesMessage` error message to generate the corresponding error messages.

For information about the `Validator` interface and `FacesMessage`, see the Javadoc for `javax.faces.validator.ValidatorException` and `javax.faces.application.FacesMessage`, or visit <http://docs.oracle.com/javaee/index.html>.

What Happens When You Create a Backing Bean Validation Method

When you create a validation method, JDeveloper adds a skeleton method to the managed bean you selected. The following example shows the code JDeveloper generates.

```
public void inputText_validator(FacesContext facesContext,
    UIComponent uiComponent, Object object) {
    // Add event code here...
}
```

When the form containing the input component is submitted, the method to which the `validator` attribute is bound is executed.

How to Create a Custom JSF Validator

Creating a custom validator requires writing the business logic for the validation by creating a `Validator` implementation of the interface, and then registering the custom validator with the application. You can also create a tag for the validator, or you can use the `f:validator` tag and the custom validator as an attribute for that tag.

You can then create a client-side version of the validator. ADF Faces client-side validation works in the same way that standard validation works on the server, except that JavaScript is used on the client. JavaScript validator objects can throw `ValidatorExceptions` exceptions and they support the `validate()` method.

Before you begin:

It may be helpful to have an understanding of custom JSF validation. See [Creating Custom JSF Validation](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for ADF Faces Converters and Validators](#).

To create a custom JSF validator:

1. Create a Java class that implements the `javax.faces.validator.Validator` interface. The implementation must contain a public no-args constructor, a set of accessor methods for any attributes, and a `validate` method to implement the `Validator` interface.

```
public void validate(FacesContext facesContext,
    UIComponent uiComponent,
    Object object)
    throws ValidatorException {
    ..
}
```

For information about these classes, refer to the Javadoc or visit <http://docs.oracle.com/javaee/index.html>.

2. Add the needed validation logic. This logic should use the `javax.faces.validate.ValidatorException` exception to throw the appropriate exceptions and the `javax.faces.application.FacesMessage` error message to generate the corresponding error messages.

For information about the `Validator` interface and `FacesMessage`, see the Javadoc for `javax.faces.validate.ValidatorException` and `javax.faces.application.FacesMessage`, or visit <http://docs.oracle.com/javaee/index.html>.

3. If your application saves state on the client, your custom validator must implement the `Serializable` interface, or the `StateHolder` interface, and the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods of the `StateHolder` interface.

See the Javadoc for the `StateHolder` interface of the `javax.faces.component` package.

4. Register the validator in the `faces-config.xml` file.

- a. Open the `faces-config.xml` file.

The `faces-config.xml` file is located in the `View_Project > WEB-INF` node in the JDeveloper Applications window.

- b. In the editor window, click the **Overview** tab.
- c. Choose **Validators** and click **Add** to add the validator information.

Click **Help** or press F1 for additional help in registering the validator.

To create a client-side version of the validator:

1. Write a JavaScript version of the validator, passing relevant information to a constructor.
2. Implement the interface `org.apache.myfaces.trinidad.validator.ClientValidator`, which has two methods. The first method is `getClientScript()`, which returns an implementation of the JavaScript `Validator` object. The second method is

`getClientValidation()`, which returns a JavaScript constructor that is used to instantiate an instance of the validator.

The following example shows a validator in Java.

```
public String getClientValidation(
    FacesContext context,
    UIComponent component)
{
    return ("new SSNValidator('Invalid social security number.', 'Value \"{1}\"
        must start with \"123\".')");
}
```

The Java validator calls the JavaScript validator shown in the following example.

```
function SSNValidator(summary, detail)
{
    this._detail = detail;
    this._summary = summary;
}
```

To use a custom validator on a JSF page:

- To use a custom validator that has a tag on a JSF page, you must manually nest it inside the component's tag.

The following example shows a custom validator tag nested inside an `inputText` component. Note that the tag attributes are used to provide the values for the validator's properties that were declared in the `faces-config.xml` file.

```
<h:inputText id="empnumber" required="true">
    <hdemo:emValidator emPatterns="9999|9 9 9 9|9-9-9-9" />
</h:inputText>
```

To use a custom validator without a custom tag:

To use a custom validator without a custom tag, nest the validator's ID (as configured in `faces-config.xml` file) inside the `f:validator` tag. The validator's ID attribute supports EL expression such that the application can dynamically determine the validator to use.

1. From the Structure window, right-click the input component for which you want to add validation, and choose **Insert inside component > JSF > Validator**.
2. With input component selected, in the Properties window, select the validator's ID from the dropdown list and click **OK**.

JDeveloper inserts code on the JSF page that makes the validator ID a property of the `f:validator` tag.

The following example shows the code on a JSF page for a validator using the `f:validator` tag.

```
<af:inputText id="empnumber" required="true">
    <f:validator validatorID="emValidator"/>
</af:inputText>
```

What Happens When You Use a Custom JSF Validator

When you use a custom JSF validator, the application accesses the validator class referenced in either the custom tag or the `f:validator` tag and executes the `validate()` method. This method executes logic against the value that is to be

validated to determine if it is valid. If the validator has attributes, those attributes are also accessed and used in the validation routine. Like standard validators, if the custom validation fails, associated messages are placed in the message queue in the `FacesContext` instance.

8

Rerendering Partial Page Content

This chapter describes how to use the partial page render features provided with ADF Faces components to rerender areas of a page without rerendering the whole page.

This chapter includes the following sections:

- [About Partial Page Rendering](#)
- [Using Partial Triggers](#)
- [Using the Target Tag to Execute PPR](#)
- [Enabling Partial Page Rendering Programmatically](#)
- [Using Partial Page Navigation](#)
- [Using a Streaming Component to Allow Page Loading](#)

About Partial Page Rendering

Oracle ADF Faces supports enabling Partial Page Rendering (PPR), which enables refreshing portions of page content without the need to redraw the entire page. You can use the `autoSubmit`, `partialSubmit`, and `partialTriggers` component attributes to enable PPR.

Ajax (Asynchronous JavaScript and XML) is a web development technique for creating interactive web applications, where web pages appear more responsive by exchanging small amounts of data with the server behind the scenes, without the whole web page being rerendered. The effect is to improve a web page's interactivity, speed, and usability.

With ADF Faces, the feature that delivers the Ajax partial page render behavior is called **partial page rendering** (PPR). During PPR, the JSF page request lifecycle (including conversion and validation) is run only for certain components on a page. Certain ADF Faces components are considered event root components, and are what determine the boundaries on which this optimized lifecycle is run.

The event root component can be determined in two ways:

- **Events:** Certain events indicate a component as a root. For example, the disclosure event sent when expanding or collapsing a `showDetail` component (see [Displaying and Hiding Contents Dynamically](#)), indicates that the `showDetail` component is a root. When the `showDetail` component is expanded or collapsed, only that component, and any of its child components, goes through the lifecycle. Other examples of events identifying a root component are the disclosure event when expanding nodes on a tree, or the sort event on a table. For a complete list of events that have corresponding event root components, see in [Events and Partial Page Rendering](#).
- **Components:** Certain components are recognized as an implicit boundary, and therefore a root component. For example, the framework knows a popup dialog is

a boundary. No matter what event is triggered inside a dialog, the lifecycle does not run on components outside the dialog. It runs only on the popup.

The following components are considered event root components:

- `popup`
- `region`
- `panelCollection`
- `calendar`
- `editableValueHolder` components (such as `inputText`)

In addition to this built-in PPR functionality, there may be cases when you want components that are outside of the boundary to be included in the optimized lifecycle. You can configure this declaratively, using the partial trigger attributes to set up dependencies so that one component acts as a trigger and another as the listener. When any event occurs on the trigger component, the lifecycle is run on the trigger and its children (as described above), and then also on any listener components and child components to the listener.

For example, suppose you have an `inputText` component on a page whose `required` attribute is set to `true`. On the same page are radio buttons that when selected cause the page to either show or hide text in an `outputText` component, as shown in [Figure 8-1](#).

Figure 8-1 Required Field and Boolean with Auto-Submit



Also assume that you want the user to be able to select a radio button before entering the required text into the field. While you could set the radio button components to automatically trigger a submit action and also set their `immediate` attribute to `true` so that they are processed before the `inputText` component, you would also have to add a `valueChangeEvent` listener, and in it, jump to the Render Response phase so that validation is not run on the input text component when the radio buttons are processed.

Instead of having to write this code in a listener, you can use the `partialTriggers` attribute to have the lifecycle run just on the radio buttons and the output text component. You would set the radio buttons to be triggers and the `panelGroupLayout` component that contains the output text to be the target, as shown in [Example 5-4](#).

 **Tip:**

Because the output text won't be rendered when it's configured to hide, it cannot be a target. Therefore it is placed in a `panelGroupLayout` component, which is then configured to be the target.

```

<af:form>
  <af:inputText label="Required Field" required="true" />
  <af:selectBooleanRadio id="show" autoSubmit="true"
text="Show"
                    value="#{validate.show}" />
  <af:selectBooleanRadio id="hide" autoSubmit="true" text="Hide"
                    value="#{validate.hide}" />
  <af:panelGroupLayout partialTriggers="show hide" id="panel">
    <af:outputText value="You can see me!" rendered="#{validate.show}" />
  </af:panelGroupLayout>
</af:form>

```

Because the `autoSubmit` attribute is set to `true` on the radio buttons, when they are selected, a `SelectionEvent` is fired, for which the radio button is considered the root. Because the `panelGroupLayout` component is set to be a target to both radio components, when that event is fired, only the `selectBooleanRadio` (the root), the `panelGroupLayout` component (the root's target), and its child component (the `outputText` component) are processed through the lifecycle. Because the `outputText` component is configured to render only when the Show radio button is selected, the user is able to select that radio button and see the output text, without having to enter text into the required input field above the radio buttons.

Note however, that when you use the partial trigger attributes to set up dependencies between components, *any* event from the trigger component will cause the target component, and its children, to execute and render. This can result in validation errors.

For example, suppose instead of using an `outputText` component in the `panelGroupLayout`, you want to use an `inputText` component whose `required` attribute is set to `true`, as shown in the following example.

```

<af:form>
  <af:selectBooleanRadio id="show1" autoSubmit="true" text="Show"
                    value="#{validate.show1}" />
  <af:selectBooleanRadio id="hide1" autoSubmit="true" text="Hide"
                    value="#{validate.hide1}" />
  <af:panelGroupLayout partialTriggers="show1 hide1">
    <af:inputText label="Required Field" required="true"
                    rendered="#{validate.show1}" />
  </af:panelGroupLayout>
</af:form>

```

In this example, the `inputText` component will be validated because the lifecycle runs on the root (the `selectBooleanRadio` component), the target (the `panelGroupLayout` component), and the target's child (the `inputText` component). Validation will fail because the `inputText` component is marked as required and there is no value, so an error will be thrown. Because of the error, the lifecycle will skip to the Render Response phase and the model value bound to the radio button will not be updated. Therefore, the `panelGroupLayout` component will not be able to show or hide because the value of the radio button will not be updated.

For cases like these, you can explicitly configure which targets you want executed and which targets you want rendered when a specific event (or events) is fired by a component. In this new example, we want the `valueChange` event on the `selectBooleanRadio` buttons to trigger PPR, but instead of executing the `panelGroupLayout` component and its children through the lifecycle, only the radio buttons should be executed. However, the entire form should be rendered. As that happens, the `inputText` component can then determine whether or not to render. When you need to explicitly determine the events that cause PPR, and the

components that should be executed and rendered, you use the `target` tag, as shown in the following example.

```
<af:panelFormLayout id="pf11">
  <af:selectBooleanRadio id="show2" autoSubmit="true" text="Show"
    value="#{validate.show2}"/>
    <af:target events="valueChange" execute="show2 hide2" render="pf11"/>
  <af:selectBooleanRadio id="hide2" autoSubmit="true" text="Hide"
    value="#{validate.hide2}"/>

    <af:target events="valueChange" execute="hide2 show2" render="pf11"/>
  <af:panelGroupLayout id="pg11">
    <af:inputText label="Required Field" required="true"
      rendered="#{validate.show2}"/>
  </af:panelGroupLayout>
</af:panelFormLayout>
```

In this example, when the `valueChange` event is fired from either of the `selectBooleanRadio` components, only the `selectBooleanRadio` components will be executed in the lifecycle, while the entire `panelFormLayout` component will be rendered.

 **Tip:**

If your application uses the Fusion technology stack, you can enable the automatic partial page rendering feature on any page. This causes any components whose values change as a result of backend business logic to be automatically rerendered. See *What You May Need to Know About Partial Page Rendering and Iterator Bindings in [Developing Fusion Web Applications with Oracle Application Development Framework](#)*.

Additionally, ADF Faces applications can use PPR for navigation. In standard JSF applications, the navigation from one page to the next requires the new page to be rendered. When using Ajax-like components, this can cause overhead because of the time needed to download the different JavaScript libraries and style sheets. To avoid this costly overhead, the ADF Faces architecture can optionally simulate full-page transitions while actually remaining on a single page, thereby avoiding the need to reload JavaScript code and skin styles.

 **Note:**

The browser must have JavaScript enabled for PPR to work.

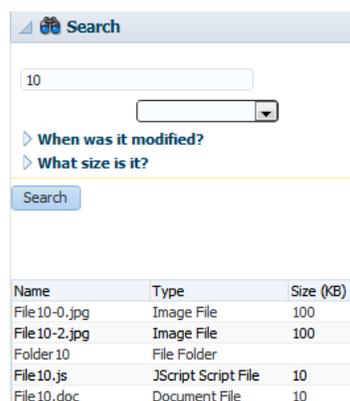
Using Partial Triggers

ADF Faces components have `partialTriggers` attribute that developers use to refresh based on the other component ID added into the `partialTrigger` attribute. Use this attribute to dynamically refresh a part of the page.

Using partial triggers, one component, referred to as the **target** component, is rerendered when any event occurs on another component, referred to as the **trigger** component.

For example, as shown in [Figure 8-2](#), the File Explorer application contains a table that shows the search results in the Search panel. This table (and only this table) is rendered when the search button is activated. The search button is configured to be the trigger and the table is configured to be the target. When any event fires from the button, the table and its components will be processed through the lifecycle (as well as the button)

Figure 8-2 The Search Button Causes Results Table to Rerender



Note:

In some cases, you may want a component to be executed or rendered only when a particular event is fired, not for every event associated with the trigger component. In these cases, you should use the target tag. See [Using the Target Tag to Execute PPR](#). When you want some logic to determine whether a component is to be executed or rendered, you can programmatically enable PPR. See [Enabling Partial Page Rendering Programmatically](#).

Trigger components must inform the framework that a PPR request has occurred. On command components, this is achieved by setting the `partialSubmit` attribute to `true`. Doing this causes the command component to fire a partial page request each time it is clicked.

For example, say a page includes an `inputText` component, a `button` component, and an `outputText` component. When the user enters a value for the `inputText` component, and then clicks the `button` component, the input value is reflected in the `outputText` component. You would set the `partialSubmit` attribute to `true` on the `button` component.

However, components other than command components can trigger PPR. ADF Faces input and select components have the ability to trigger partial page requests automatically whenever their values change. To make use of this functionality, use the `autoSubmit` attribute of the input or select component so that as soon as a value is

entered, a submit occurs, which in turn causes a `valueChangeEvent` event to occur. It is this event that notifies the framework to execute a PPR, as long as a target component is set. In the previous example, you could delete the `button` component and instead set the `inputText` component's `autoSubmit` attribute to `true`. Each time the value changes, a PPR request will be fired.

 **Tip:**

The `autoSubmit` attribute on an input component and the `partialSubmit` attribute on a command component are not the same thing. When `partialSubmit` is set to `true`, then only the components that have values for their `partialTriggers` attribute will be processed through the lifecycle.

The `autoSubmit` attribute is used by input and select components to tell the framework to automatically do a form submit whenever the value changes. When a form is submitted and the `autoSubmit` attribute is set to `true`, a `valueChangeEvent` event is invoked, and the lifecycle runs only on the components marked as root components for that event, and their children. See [Using the Optimized Lifecycle](#).

Once PPR is triggered, any component configured to be a target will be processed through the lifecycle. You configure a component to be a target by setting the `partialTriggers` attribute to the relative ID of the trigger component. For information about relative IDs, see [Locating a Client Component on a Page](#).

In the example, to update the `outputText` in response to changes to the `inputText` component, you would set its `partialTriggers` attribute to the `inputText` component's relative ID.

Note that certain events on components trigger PPR by default, for example the `disclosure` event on the `showDetail` component and the `sort` event on a table. This means that any component configured to be a target by having its `partialTriggers` attribute set to that component's ID will rerender when these types of events occur. When you don't want all events to trigger PPR, then instead of using the `partialTriggers` attribute, you should use the `target` tag. This tag allows you to explicitly set which events will cause PPR.

Another example of when to use the `target` tag instead of the `partialTriggers` attribute is when your trigger component is an `inputLov` or an `inputComboBoxLov`, and the target component is a dependent input component set to `required`. In this case, a validation error will be thrown for the input component when the LOV popup is displayed. If you use the `target` tag instead, you can explicitly set which components should execute (the LOV), and which should be rendered (the input component). See [Using the Target Tag to Execute PPR](#).

How to Use Partial Triggers

For a component to be rendered based on an event caused by another component, it must declare which other components are the triggers.

 **Best Practice:**

Do not use both partial triggers and the `target` tag on the same page. When in doubt, use only the `target` tag. See [Using the Target Tag to Execute PPR](#).

Before you begin:

It may be helpful to have an understanding of declarative partial page rendering. See [Using Partial Triggers](#).

To use partial triggers:

1. In the Structure window, select the trigger component (that is, the component whose action will cause the PPR):
 - Expand the **Common** section of the Properties window and set the `id` attribute if it is not already set. Note that the value must be unique within that component's naming container. If the component is not within a naming container, then the ID must be unique to the page. For information about naming containers, see [Locating a Client Component on a Page](#).

 **Tip:**

JDeveloper automatically assigns component IDs. You can safely change this value. A component's ID must be a valid XML name, that is, you cannot use leading numeric values or spaces in the ID. JSF also does not permit colons (:) in the ID.

- If the trigger component is a command component, expand the **Behavior** section of the Properties window, and set the `partialSubmit` attribute to `true`.
- If the trigger component is an input or select component in a form and you want the value to be submitted, expand the **Behavior** section of the Properties window, and set the `autoSubmit` attribute of the component to `true`.

 **Note:**

Set the `autoSubmit` attribute to `true` only if you want the component to submit its value. If you do not want to submit the value, then some other logic must cause the component to issue a `ValueChangeEvent` event. That event will cause PPR by default and any component that has the trigger component as its value for the `partialTriggers` attribute will be rerendered.

2. In the Structure window, select the target component that you want to rerender when a PPR-triggering event takes place.
3. Expand the **Behavior** section of the Properties window, click the icon that appears when you hover over the `partialTriggers` attribute and choose **Edit**.

4. In the Edit Property dialog, shuttle the trigger component to the **Selected** panel and click **OK**. If the trigger component is within a naming container, JDeveloper automatically creates the relative path for you.

 **Tip:**

The `selectBooleanRadio` components behave like a single component with partial page rendering; however, they are in fact multiple components. Therefore, if you want other components (such as `inputText` components) to change based on selecting a different `selectBooleanRadio` component in a group, you must group them within a parent component, and set the `partialTriggers` attribute of the parent component to point to all of the `SelectBooleanRadio` components.

The following example shows a `link` component configured to execute PPR.

```
<af:link id="deleteFromCart" partialSubmit="true"
        actionListener="#{homeBean...}" text="Delete From Cart">
```

The following example shows an `outputText` component that will be rerendered when the link with ID `deleteFromCart` in the previous example is clicked.

```
<af:outputText id="estimatedTotalInPopup"
               partialTriggers="deleteFromCart"
               value="#{shoppingCartBean...}"/>
```

 **Tip:**

If you need to prevent components from being validated on a page during PPR, then you should use the `target` tag. See [Using the Target Tag to Execute PPR](#).

What You May Need to Know About Using the Browser Back Button

In an ADF Faces application, because some components use PPR (either implicitly or because they have been configured to listen for a partial trigger), what happens when a user clicks the browser's back button is slightly different than in an application that uses simple JSF components.

In an application that uses simple JSF components, when the user clicks the browser's back button, the browser returns the page to the state of the DOM (document object model) as it was when last rendered, but the state of the JavaScript is as it was when the user first entered the page.

For example, suppose a user visited PageA. After the user interacts with components on the page, say a PPR event took place using JavaScript. Let's call this new version of the page PageA1. Next, say the user navigates to PageB, then clicks the browser back button to return to PageA. The user will be shown the DOM as it was on PageA1, but the JavaScript will not have run, and therefore parts of the page will be as they were for PageA. This might mean that changes to the page will be lost. Refreshing the page will run the JavaScript and so return the user to the state it was in PageA1. In an

application that uses ADF Faces, the refresh is not needed; the framework provides built-in support so that the JavaScript is run when the back button is clicked.

What You May Need to Know About PPR and Screen Readers

Screen readers do not reread the full page in a partial page request. PPR causes the screen reader to read the page starting from the component that fired the partial page request. You should place the target components after the component that triggers the partial request; otherwise, the screen reader would not read the updated target components.

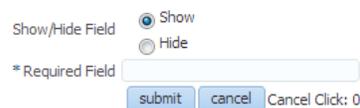
Using the Target Tag to Execute PPR

ADF Faces provides the `target` tag that provides a declarative way to allow a component to specify the list of targets it wants executed and rendered when an event is fired by the component. It has complete control on which components the JSF lifecycle is executed, and which components are rerendered (refreshed).

For components such as tables that have many associated events, PPR will happen each time an event is triggered, causing any child component of the table, or any component with the table as a partial trigger, to be executed and rendered. If you want components to be rendered or executed only for certain events, or if you only want certain components to execute or render, you can use the `target` tag.

Using the target tag can be especially useful when you want to skip component validation under very specific circumstances. For example, say you have a form with a required field, along with a **Submit** button and a **Cancel** button, as shown in [Figure 8-3](#).

Figure 8-3 Required Field is Processed When Cancel Button is Clicked



Under normal circumstances, when the **Cancel** button is clicked, all fields in the form are processed. Because the input text field is required, it will fail validation.

To avoid this failure, you can use the `target` tag as a child to the **Cancel** button. Using the `target` tag, you can state which targets you want executed and rendered when a specific event (or events) is fired by the component. In this example, you might configure the **Cancel** button's target tag so that only that button is executed, as shown in the following example.

```
<af:panelFormLayout id="pf11" labelAlignment="top">
  .
  .
  .
  <af:panelLabelAndMessage id="plam12" for="it3">
    <af:inputText label="Required Field" required="true"
      rendered="{validate.show3}" id="it3"/>
  </af:panelLabelAndMessage>
  <af:panelGroupLayout layout="horizontal" id="pg19">
    <af:button text="submit" partialSubmit="true" id="cb2"
      <af:target execute="@this it3" render="pf11"/>
    </af:button>
  </af:panelGroupLayout>
</af:panelFormLayout>
```

```

<af:button actionListener="#{validate.handleCancel}"
           partialSubmit="true" text="cancel" id="cb1">
  <af:target execute="@this"/>
</af:button>
</af:panelGroupLayout>
</af:panelFormLayout>

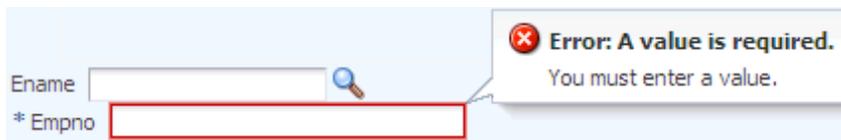
```

In this example, when the Submit button is clicked, that button and the `inputText` component are executed, and the contents of the `panelFormLayout` component are rendered. However, when the Cancel button is clicked, only the Cancel button is executed. Nothing is rendered as a result of the click.

Another common validation issue can occur when you have an LOV component and a dependent input text component. For example, say you have an `inputListOfValues` component from which the user selects an employee name. You want the Employee Number input text field to be automatically populated once the user selects the name from the LOV, and you also want this field to be required.

If you were to use partial triggers by setting the LOV as the trigger, and the input text component as a target, the input text component would fail validation when the popup closes. When the user clicks the search icon, the `inputText` component will be validated because the lifecycle runs on both the root (the `inputListOfValues` component) and the target (the `inputText` component). Validation will fail because the `inputText` component is marked as required and there is no value, as shown in [Figure 8-4](#)

Figure 8-4 Validation Error is Thrown Because a Value is Required



Instead, you can use a `target` tag as a child to the `inputListOfValues` component, and configure it so that only the `inputListOfValues` component is executed, but the input text component is rendered with the newly selected value, as shown in the following example.

```

<af:panelFormLayout id="pf12">
  <af:inputListOfValues label="Ename" id="lov21" required="true"
                       value="#{validateLOV.ename}" autoSubmit="true"
                       popupTitle="Search and Select: Ename" searchDesc="Choose a name"
                       model="#{validateLOV.listOfValuesModel}"
                       validator="#{validateLOV.validate}">
    <af:target execute="@this" render="it1"/>
  </af:inputListOfValues>
  <af:inputText label="Empno" value="#{validateLOV.empno}" required="true"
               id="it1"/>
</af:panelFormLayout>

```

How to Use the Target Tag to Enable Partial Page Rendering

You place the `target` tag as a child to the component whose event will cause the PPR. For example, for the form shown in [Figure 8-3](#), you would place the target tag as

a child to the **Cancel** button. You then explicitly name the events and components to participate in PPR.

 **Best Practice:**

Do not use both partial triggers and the `target` tag on the same page. When in doubt, use only the `target` tag.

Before you begin:

It may be helpful to have an understanding of form components. See [Using the Target Tag to Execute PPR](#).

To use the target tag:

1. In the Components window, from the Operations panel, drag a **target** and drop it as a child to the component that triggers events for which you want to control the PPR.
2. In the Properties window, set the following:
 - **Events:** Enter a space delimited list of events that you want the target tag to handle for the component. The following events are supported:
 - action
 - calendar
 - calendarActivity
 - calendarActivity
 - durationChange
 - calendarDisplayChange
 - carouselSpin
 - contextInfo
 - dialog
 - disclosure
 - focus
 - item
 - launch
 - launchPopup
 - poll
 - popupCanceled
 - popupFetch
 - query
 - queryOperation
 - rangeChange

- regionNavigation
- return
- returnPopupData
- returnPopup
- rowDisclosure
- selection
- sort
- valueChange

Any events that the component executes that you do not list will be handled as usual by the framework.

You can also use the `@all` keyword if you want the target to control all events for the component.

The default is `@all`.

- **Execute:** Enter a space delimited list of component IDs for the components that should be executed in response to the event(s) listed for the `event` attribute. Instead of component IDs, you can use the following keywords:
 - `@this`: Only the parent component of the target will be executed.
 - `@all`: All components within the same `af:form` tag will be executed. Using this means that in effect, no PPR will occur, as all components will execute as in a normal page lifecycle.
 - `@default`: Execution of components will be handled using event root components or partial triggers, in other words, as if no target tag were used. This value is the default.

**Note:**

While the JSF `f:ajax` tag supports the `@none` keyword, the `af:target` tag does not.

- **Render:** Enter a space delimited list of component IDs for the components that should be rendered in response to the event(s) listed for the `event` attribute. Instead of component IDs, you can use the following keywords:
 - `@this`: Only the parent component of the target will be rendered.
 - `@all`: All components within the same `af:form` tag will be rendered.
 - `@default`: Rendering of components will be determined by the framework. This value is the default.

The following example shows the code for the form with the radio buttons and required field shown in [Figure 8-3](#). For the first target tag, because it's a child to the `show3` radio button, and it is configured to execute the `show3` and `hide3` radio buttons and render the `panelFormLayout`, when that button is selected, only the two radio buttons are run through the lifecycle and the entire form is rendered. The `inputText` component is not executed because it is not in the `execute` list, but it is rendered because it is included in the form.

The `target` tag is also used on the **Submit** and **Cancel** buttons. For the **Submit** button, because the input text component should be processed through the lifecycle, it is listed along with the **Submit** button as values for its `target`'s `execute` attribute. However, for the **Cancel** button, because the input text component should not be processed, it is not listed for that `target`'s `execute` attribute, and so will not throw a validation error.

```
<af:panelFormLayout id="pf11">
  <af:panelLabelAndMessage label="Show/Hide Field" id="plam11">
    <af:selectBooleanRadio id="show3" autoSubmit="true" text="Show"
group="group3"
                        value="{validate.show3}">
      <af:target execute="show3 hide3" render="pf11"/>
    </af:selectBooleanRadio>
    <af:selectBooleanRadio id="hide3" autoSubmit="true" text="Hide"
group="group3"
                        value="{validate.hide3}">
      <af:target execute="hide3 show3" render="pf11"/>
    </af:selectBooleanRadio>
  </af:panelLabelAndMessage>
  <af:panelLabelAndMessage label="Required Field" id="plam12"
showRequired="true">
    <af:inputText label="Required Field" required="true" simple="true"
rendered="{validate.show3}" id="it3"/>
  </af:panelLabelAndMessage>
  <af:panelGroupLayout layout="horizontal" id="pg19">
    <f:facet name="separator">
      <af:spacer width="2px" id="s7"/>
    </f:facet>

    <af:commandButton text="submit" partialSubmit="true"
disabled="{validate.hide3}" id="cb2">
      <af:target execute="@this it3" render="pf11"/>
    </af:commandButton>
    <af:commandButton actionListener="{validate.handleCancel}"
partialSubmit="true" text="cancel" id="cb1">
      <af:target execute="@this" render="ot10"/>
    </af:commandButton>

    <af:outputText clientComponent="true"
value="Cancel Click: {validate.clickCount}" id="ot10"/>
  </af:panelGroupLayout>
```

Enabling Partial Page Rendering Programmatically

The partial page rendering feature of ADF Faces can be implemented programmatically with the help of certain framework objects, such as `FaceContext` and `AdfFacesContext`. Use the `addPartialTarget` method to enable partial page rendering programmatically.

When you want a target to be rerendered based on specific logic, you can enable partial page rendering programmatically.

How to Enable Partial Page Rendering Programmatically

You use the `addPartialTarget` method to enable partial page rendering.

How to enable PPR programmatically:

1. Create a listener method for the event that should cause the target component to be rerendered.

Use the `addPartialTarget()` method to add the component (using its ID) as a partial target for an event, so that when that event is triggered, the partial target component is rerendered. Using this method associates the component you want to have rerendered with the event that is to trigger the rerendering.

For example, the File Explorer application contains the `NavigatorManager.refresh()` method. When invoked, the navigator accordion is rerendered.

```
public void refresh()
{
    for (BaseNavigatorView nav: getNavigators())
    {
        nav.refresh();
    }

    AdfFacesContext adfFacesContext = AdfFacesContext.getCurrentInstance();
    adfFacesContext.addPartialTarget(_navigatorAccordion);
}
```

2. In the JSF page, select the target component. In the Properties window, enter a component ID and set **ClientComponent** to **true**.

 **Note:**

You must set the `clientComponent` attribute to `true` to ensure that a client ID will be generated.

3. In the Properties window, find the listener for the event that will cause the refresh and bind it to the listener method created in Step 1.

Using Partial Page Navigation

In ADF Faces, navigation between pages can be made much faster by using Ajax technologies to call the new page, which skips the initialization steps. To use this partial page navigation feature, you must first enable partial page rendering.

Instead of performing a full page transition in the traditional way, you can configure an ADF Faces application to have navigation triggered through a PPR request. The new page is sent to the client using PPR. Partial page navigation is disabled by default.

When partial page navigation is used, in order to keep track of location (for example, for bookmarking purposes, or when a refresh occurs), the framework makes use of the hash portion of the URL. This portion of the URL contains the actual page being displayed in the browser.

Additionally, JavaScript and CSS will not be loaded for each page. You must use the `resource` tag to include JavaScript and CSS content specific to the current page. Using the `<f:verbatim>` or `<trh:stylesheet>` tags will not work. See [Adding JavaScript to a Page](#).

When partial page navigation is enabled in an application, `get` requests are supported for the button and link components:

 **Note:**

PPR `get` requests are not supported in Internet Explorer. When using that browser, URLs will be loaded using a standard `get` request.

For other browsers, `get` requests for these components are only supported for pages within an application.

How to Use Partial Page Navigation

You can turn partial page navigation on by setting the `oracle.adf.view.rich.pprNavigation.OPTIONS` context parameter in the `web.xml` file to `on`.

Before you begin:

It may be helpful to have an understanding of partial page navigation. See [Using Partial Page Navigation](#).

To use partial page navigation:

1. Double-click the `web.xml` file.
2. In the source editor, change the `oracle.adf.view.rich.pprNavigation.OPTIONS` parameter to one of the following:
 - `on`: Enables partial page navigation.

 **Note:**

If you set the parameter to `on`, then you need to set the `partialSubmit` attribute to `true` for any command components involved in navigation.

- `onWithForcePPR`: Enables partial page navigation and notifies the framework to use the PPR channel for all action events, even those that do not result in navigation. Since partial page navigation requires that the action event be sent over PPR channel, use this option to easily enable partial page navigation.

When partial page navigation is used, normally only the visual contents of the page are re-rendered (the header content remains constant for all pages). However, the entire document will be re-rendered when an action on the page is defined to use full page submit and also when an action does not result in navigation.

What You May Need to Know About PPR Navigation

Before using PPR navigation, you should be aware of the following:

- When using PPR navigation, all pages involved in this navigation must use the same CSS skin.

- You must use the `resource` tag to include JavaScript and CSS content specific to the current page.
- Unlike regular page navigation, partial navigation will not result in JavaScript globals (variables and functions defined in global scope) being unloaded. This happens because the window object survives partial page transition. Applications wishing to use page-specific global variables and/or functions must use the `AdfPage.getPageProperty()` and `AdfPage.setPageProperty()` methods to store these objects.

Using a Streaming Component to Allow Page Loading

ADF Faces supports a streaming component in a JSF page that allows components to delay their rendering until a streaming request is fulfilled. A placeholder is displayed until the data is fetched from the data source.

The streaming component lets the page continue to load while database queries are being run. It utilizes NOSCRIPT components which are non-element based components that support partial page rendering and data streaming. You can use this component for infolets in a JSF page. An infolet is small region within a JSF page that displays the data fetched from different data sources. Streaming components can be used for infolets in a JSF page or for the individual components within an infolet of the JSF page.

When there is a streaming request by the client for an infolet or a component, the streaming framework launches background threads for a data model so that the data is fetched in parallel from different data sources. During the data fetch, the streaming component renders a placeholder on the screen in the initial rendering and the actual content replaces the placeholder when it becomes available. This placeholder can be a text or an image. You can use the `image` component or `outputText` component for the placeholder to display an image or to display text during the process of data fetch.

How to Implement an Instance of AsyncFetch to Fetch Streaming Data

The ADF Faces streaming component implements the `AsyncFetch` interface to fetch the data asynchronously on a streaming request. `AsyncFetch` is an interface to be implemented by the data models that support fetching data on the thread other than the request thread.

To use a streaming component, you must first configure a managed bean in `adfc-config.xml` file. In the below example, `streamingBean` is configured as a managed bean.

A streaming component contains a value attribute that accepts an implementation class instance of `oracle.adf.view.rich.model.AsyncFetch`. `AsyncFetch` is an interface that supports fetching data on threads. The component will query `AsyncFetch` class to see if the data is available on the first rendering. If it is not, a placeholder will be rendered. This can be achieved by a `placeholder` streaming facet. A streaming facet contains the placeholder components, typically text or graphics. The example below shows how an image with a short description is added to a streaming facet. Once the data is fetched during the streaming request, the streaming framework rerenders the component first and then the children are rendered, replacing the streaming facet on the client with the children content.

The following example shows the JSF page code for streaming.

```
<af:streaming id="s4" value="#{streamingBean.dataStore4}">
  <f:facet name="placeholder">
    <?audit suppress oracle.jdeveloper.jsp.check-valid-parent?>
    <af:image id="placeholder4" source="/images/streaming-load.gif"
shortDesc="Please wait..." inlineStyle="margin: 4px;"/>
  </f:facet>
  <af:panelHeader text="Links" size="5" id="ph4">
    <af:panelGroupLayout id="pgl5">
      <af:iterator value="#{streamingBean.streamingData4.data}"
var="link" id="i4">
        <af:goLink text="#{link.name}" id="l1"
destination="#{link.url}" targetFrame="link" styleClass="singleRow"/>
      </af:iterator>
    </af:panelGroupLayout>
  </af:panelHeader>
</af:streaming>
```

In the above example, `streamingBean` is configured as a managed bean. The `streamingBean.dataStore4` value attribute represents the content fetched from the data source to be rendered. The `placeholder` streaming facet contains the `streaming-load.gif` image along with a short description of `Please wait...`. The `dataStore4` get method of `streamingBean` returns an implementation class instance of `oracle.adf.view.rich.model.AsyncFetch`.

 **Note:**

You can also specify the data to be fetched by the model by providing an EL expression as a fetch constraint for a streaming component. This EL expression is passed to the `AsyncFetcher` methods, which may utilize the expression as a fetch constraint. See *Java API Reference for Oracle ADF Faces* for more information on `AsyncFetch` and `AsyncFetcher` methods.

How to Create a Streaming Component

A streaming component permits a component to be rendered in a page during a streaming request so that the initial page rendering is not delayed when a model is expected to take a longer time to retrieve its data.

Before you begin:

To use a streaming component, you must first configure a managed bean in `adfc-config.xml` file. See [Creating and Using Managed Beans](#) for information on how to create and configure a managed bean.

To create a streaming component:

1. In the Components window, from the ADF Faces panel, drag and drop **Streaming** onto the JSF page.
2. In the Properties window, expand the Other section, and set the following:

- **Constraints:** Specify the data to be fetched by the model by providing an EL expression as a fetch constraint for the streaming component. The EL expression specified for this attribute is passed to the `AsyncFetcher` methods, which may utilize the expression as a fetch constraint.
- **Rendered:** You can configure whether the component must be rendered or not using the `rendered` attribute. If the rendered attribute is set to false, no output will be delivered.
- **Value:** Displays the content fetched from the data source to be rendered.

See *Tag Reference for Oracle ADF Faces* for information on configuring different attributes for a streaming component.

Part III

Creating Your Layout

Part III describes how to use the layout components to design a page, and then how to create page fragments, templates, and how to reuse them in an application.

Specifically, this part contains the following chapters:

- [Organizing Content on Web Pages](#)
- [Creating and Reusing Fragments, Page Templates, and Components](#)

9

Organizing Content on Web Pages

This chapter describes how to use several of the ADF Faces layout components to organize content on web pages.

This chapter includes the following sections:

- [About Organizing Content on Web Pages](#)
- [Starting to Lay Out a Page](#)
- [Arranging Content in a Grid](#)
- [Displaying Contents in a Dynamic Grid Using a masonryLayout Component](#)
- [Achieving Responsive Behavior Using matchMediaBehavior Tag](#)
- [Arranging Contents to Stretch Across a Page](#)
- [Using Splitters to Create Resizable Panes](#)
- [Arranging Page Contents in Predefined Fixed Areas](#)
- [Arranging Content in Forms](#)
- [Arranging Contents in a Dashboard](#)
- [Displaying and Hiding Contents Dynamically](#)
- [Displaying or Hiding Contents in Panels](#)
- [Adding a Transition Between Components](#)
- [Displaying Items in a Static Box](#)
- [Displaying a Bulleted List in One or More Columns](#)
- [Grouping Related Items](#)
- [Separating Content Using Blank Space or Lines](#)

About Organizing Content on Web Pages

ADF provides the Alta skin by default, that is based on UI design principles to create a responsive web page. You can also use a skin other than Alta and create a template that you can use to design the layout of pages.

A new web application that you create in this release uses the Alta skin by default. To get the full benefit of the Oracle Alta UI system, Oracle recommends that you go beyond simply using the Alta skin and design your application around the Oracle Alta UI Design Principles. Designing your application using these principles enables you to make use of the layouts, responsive designs and components the Oracle Alta UI system incorporates to present content to your end users in a clean and uncluttered way. For information about the Oracle Alta UI system and the Oracle Alta UI Design Principles, see <http://www.oracle.com/webfolder/ux/middleware/alta/index.html> and for information about Oracle Alta UI Patters, see <http://www.oracle.com/webfolder/ux/middleware/alta/patterns/index.html>.

If you are using a skin other than Alta, ADF Faces provides a number of layout components that can be used to arrange other components on a page. Usually, you begin building your page with these components. You then add components that provide other functionality (for example rendering data or rendering buttons) either inside facets or as child components to these layout components.

 **Tip:**

You can create page templates that allow you to design the layout of pages in your application. The templates can then be used by all pages in your application. See [Creating and Reusing Fragments, Page Templates, and Components](#).

In addition to layout components that simply act as containers, ADF Faces also provides interactive layout components that can display or hide their content, that provide transitions between its child components, or that provide sections, lists, or empty space. Some layout components also provide geometry management functionality, such as stretching their contents to fit the browser windows as the window is resized, or the capability to be stretched when placed inside a component that stretches. [ADF Faces Layout Components](#) describes each of the layout components and their associated geometry management capabilities. For information about stretching and other geometry management functionality of layout components, see [Geometry Management and Component Stretching](#).

ADF Faces Layout Components

[Table 9-1](#) briefly describes each of the ADF Faces layout components.

Table 9-1 ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
Page Management Components			
document	Creates each of the standard root elements of an HTML page: <html>, <body>, and <head>. All pages must contain this component. See Starting to Lay Out a Page .	X	
form	Creates an HTML <form> element. See Starting to Lay Out a Page .		
Page Layout Containers			
panelGridLayout	Used in conjunction with <code>gridRow</code> and <code>gridCell</code> components to provide an HTML table-like layout where you define the rows and cells, and then place other components as children to the cells. See Arranging Content in a Grid .	X (when the <code>gridRow</code> and <code>gridCell</code> components are configured to stretch)	X (when the <code>dimensionsFrom</code> attribute is set to parent)

Table 9-1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
panelStretchLayout	Contains top, bottom, start, center, and end facets where you can place other components. See Arranging Contents to Stretch Across a Page .	X	X (when the dimensionsFrom attribute is set to parent)
panelSplitter	Divides a region into two parts (first facet and second facet) with a repositionable divider between the two. You can place other components within the facets. See Using Splitters to Create Resizable Panes .	X	X (when the dimensionsFrom attribute is set to parent)
panelDashboard	Provides a columnar display of child components (usually panelBox components). See Arranging Contents in a Dashboard .	X	X (when the dimensionsFrom attribute is set to parent)
masonryLayout	Provides a dynamically-sized grid of child components. See Displaying Content in a Dynamic Grid Using a masonryLayout Component		X
panelBorderLayout	Can have child components, which are placed in its center, and also contains 12 facets along the border where additional components can be placed. These will surround the center. See Arranging Page Contents in Predefined Fixed Areas .		
panelFormLayout	Positions input form controls, such as inputText components so that their labels and fields line up vertically. It supports multiple columns, and contains a footer facet. See Arranging Content in Forms .		
Components with Show/Hide Capabilities			
showDetailHeader	Can hide or display contents below the header. Often used as a child to the panelHeader component. See Displaying and Hiding Contents Dynamically .	X (if the type attribute is set to stretch)	X (if the type attribute is set to stretch)
showDetailItem	Used to hold the content for the different panes of the panelAccordion or different tabs of the panelTabbed component. See Displaying or Hiding Contents in Panels .	X (if it contains a single child component and its stretchChildren attribute is set to first.)	
panelBox	Titled box that can contain child components. Has a toolbar facet. See Displaying and Hiding Contents Dynamically .	X (if it is being stretched or if the type attribute is set to stretch)	X

Table 9-1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
panelAccordion	Used in conjunction with <code>showDetailItem</code> components to display as a panel that can be expanded or collapsed. See Displaying or Hiding Contents in Panels .		X (when the <code>dimensionsFrom</code> attribute is set to parent)
panelTabbed	Used in conjunction with <code>showDetailItem</code> components to display as a set of tabbed panels. See Displaying or Hiding Contents in Panels . If you want the tabs to be used in conjunction with navigational hierarchy, for example each tab is a different page or region that contains another set of navigation items, you may instead want to use a <code>navigationPane</code> component in a navigational menu. See Using Navigation Items for a Page Hierarchy .		X (when the <code>dimensionsFrom</code> attribute is set to parent)
panelDrawer	Used in conjunction with <code>showDetailItem</code> components to display as a set of tabs that can open and close like a drawer. See Displaying or Hiding Contents in Panels .		X
panelSpringboard	Used in conjunction with <code>showDetailItem</code> components to display as a set of icons, either in a grid or in a strip. When the user clicks an icon, the associated <code>showDetailItem</code> contents display below the strip. See Displaying or Hiding Contents in Panels .		X
showDetail	Hides or displays content through a toggle icon. See Displaying and Hiding Contents Dynamically .		
Miscellaneous Containers			
deck	Provides animated transitions between its child components, using the <code>af:transition</code> tag. See Adding a Transition Between Components .	X	X
panelHeader	Contains child components and provides a header that can include messages, toolbars, and help topics. See Displaying Items in a Static Box .	X (if the <code>type</code> attribute is set to <code>stretch</code>)	X (if the <code>type</code> attribute is set to <code>stretch</code>)
panelCollection	Used in conjunction with collection components such as <code>table</code> , <code>tree</code> and <code>treeTable</code> to provide menus, toolbars, and status bars for those components. See Displaying Table Menus, Toolbars, and Status Bars .	X (only a single table, tree, or tree table)	X

Table 9-1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
decorativeBox	Creates a container component whose facets use style themes to apply a bordered look to its children. This component is typically used as a container for the <code>navigationPane</code> component that is configured to display tabs. See Using Navigation Items for a Page Hierarchy .	X (in the Center facet)	X (when the <code>dimensionsFrom</code> attribute is set to parent)
inlineFrame	Creates an inline <code>iframe</code> tag.		X
navigationPane	Creates a series of navigation items representing one level in a navigation hierarchy. See Using Navigation Items for a Page Hierarchy .		X (if configured to display tabs)
panelList	Renders each child component as a list item and renders a bullet next to it. Can be nested to create hierarchical lists. See Displaying a Bulleted List in One or More Columns .		
panelWindow	Displays child components inside a popup window. See Declaratively Creating Popups .		
toolbox	Displays child toolbar and menu components together. See Using Toolbars .		
Grouping Containers			
panelGroupLayout	Groups child components either vertically or horizontally. For JSP pages, used in facets when more than one component is to be contained in a facet (Facelet pages can handle multiple children in a facet). See Grouping Related Items .		X (only if set to scroll or vertical layout)
group	Groups child components without regard to layout unless handled by the parent component of the group. For JSP pages, used in facets when more than one component is to be contained in a facet (Facelet pages can handle multiple children in a facet). See Grouping Related Items .		
Spacing Components			
separator	Creates a horizontal line between items. See Separating Content Using Blank Space or Lines .		
spacer	Creates an area of blank space. See Separating Content Using Blank Space or Lines .		

Additional Functionality for Layout Components

Once you have added a layout component to your page, you may find that you need to add functionality such as responding to events. Following are links to other functionality that layout components can use.

- **Templates:** Once you create a layout, you can save it as a template. When you make layout modifications to the template, all pages that consume the template will automatically reflect the layout changes. See [Using Page Templates](#).
- **Themes:** Themes add color styling to some of layout components, such as the `panelBox` component. For information about themes, see [Customizing the Appearance Using Styles and Skins](#)
- **Skins:** You can change the icons and other properties of layout components using skins. See [Customizing the Appearance Using Styles and Skins](#).
- **Localization:** Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- **Accessibility:** You can make your input components accessible. See [Developing Accessible ADF Faces Pages](#).
- **Using parameters in text:** You can use the ADF Faces EL format tags if you want text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Events:** Layout components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- **User customization:** Some of the components have areas that can be expanded or collapsed, such as the `showDetailHeader` component. You can configure your application so that the state of the component (expanded or collapsed) can be saved when the user leaves the page. See [Allowing User Customization on JSF Pages](#).

Starting to Lay Out a Page

When you layout a page, all the ADF faces components should be enclosed within the document, so that the document tag creates the root elements for the client page at runtime. Additionally, the document tag allows the capable components to stretch to fill the available browser space.

JSF pages that use ADF Faces components must have the `document` tag enclosed within a `view` tag. All other components that make up the page then go in between `<af:document>` and `</af:document>`. The `document` tag is responsible for rendering the browser title text, as well as the invisible page infrastructure that allows other components in the page to be displayed. For example, at runtime, the `document` tag creates the root elements for the client page. In HTML output, the standard root elements of an HTML page, namely, `<html>`, `<head>`, and `<body>`, are generated.

By default, the `document` tag is configured to allow capable components to stretch to fill available browser space. You can further configure the tag to allow a specific component to have focus when the page is rendered, or to provide messages for failed connections or warnings about navigating before data is submitted. See [How to Configure the document Tag](#).

Typically, the next component used is the ADF Faces `form` component. This component creates an HTML `form` element that can contain controls that allow a user to interact with the data on the page.

 **Note:**

Even though you can have multiple HTML forms on a page, you should have only a single ADF Faces `form` tag per page. See [Defining Forms](#).

JDeveloper automatically inserts the `view`, `document`, and `form` tags for you, as shown in the following example. See [Creating a View Page](#).

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html>
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <af:document title="untitled1.jsf" id="d1">
    <af:form id="f1"></af:form>
  </af:document>
</f:view>
```

Once those tags are placed in the page, you can use the layout components to control how and where other components on the page will render. The component that will hold all other components is considered the **root** component. Which component you choose to use as the root component depends on whether you want the contained components to display their contents so that they stretch to fit the browser window, or whether you want the contents to flow, using a scrollbar to access any content that may not fit in the window. For information about stretching and flowing, see [Geometry Management and Component Stretching](#).

 **Tip:**

Instead of creating your layout yourself, you can use JDeveloper's quick layout templates, which provide correctly configured components that will display your page with the layout you want. See [Using Quick Start Layouts](#).

Geometry Management and Component Stretching

Geometry management is the process by which the user, parent components, and child components negotiate the actual sizes and locations of the components in an application. For example, a component might be resized when it's first loaded into a browser, when the browser is resized, or when a user explicitly resizes it.

By default, if there is only a single effective visual root component, that root component will stretch automatically to consume the browser's viewable area, provided that component supports geometry management. Examples of geometry management components are `panelGridLayout` and `panelSplitter`. If the root component supports stretching its child components (and they in turn support being stretched), the size of the child components will also recompute, and so on down the component hierarchy until a flowing layout area is reached; that is, an area that does

not support stretching of its child components. You do not have to write any code to enable the stretching.



Note:

The framework does not consider popup dialogs, popup windows, or non-inline messages as root components. If a `form` component is the direct child component of the `document` component, the framework will look inside the `form` tag for the visual root. For information on sizing a popup, see [Using Popup Dialogs, Menus, and Windows](#).

As shown in [Table 9-1](#), the `panelGridLayout`, `panelStretchLayout`, `panelSplitter`, and `panelDashboard` components are components that can be stretched and can also stretch their child components. Additionally, when the `showDetailItem` component is used as a direct child of the `panelAccordion` or `panelTabbed` component, the contents in the `showDetailItem` component can be stretched. Therefore, the `panelStretchLayout`, `panelSplitter`, `panelDashboard`, `panelAccordion` with a `showDetailItem` component, and a `panelTabbed` with a `showDetailItem` component, are the components you should use as root components when you want to make the contents of the page fill the browser window.

For example, [Figure 9-1](#) shows a table placed in the center facet of the `panelStretchLayout` component. The table stretches to fill the browser space. When the entire table does not fit in the browser window, scrollbars are added in the data body section of the table.

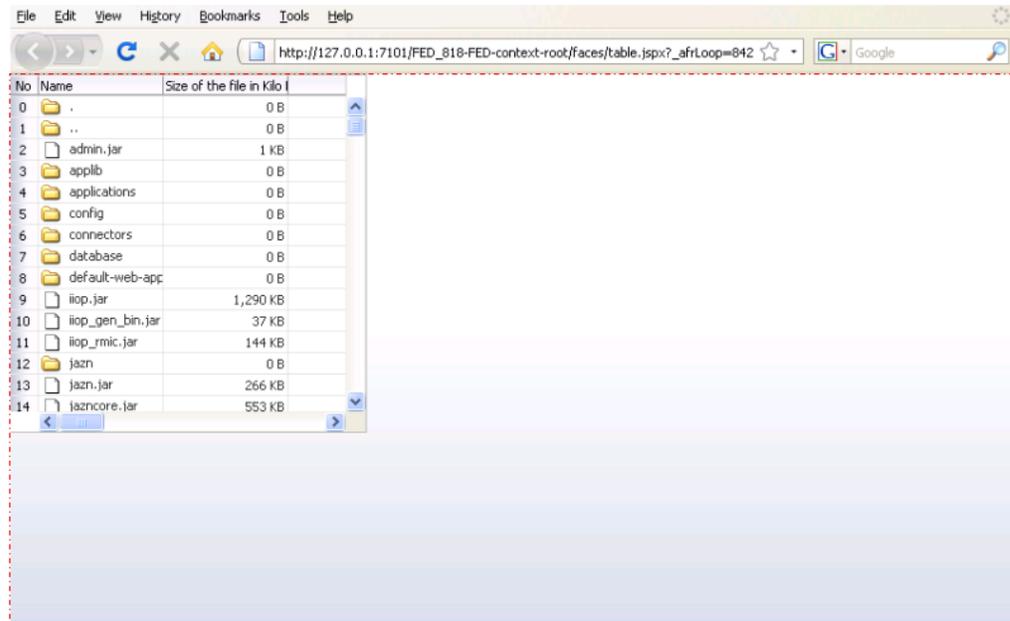
Figure 9-1 Table Inside a Component That Stretches Child Components

No	Name	Size of the file in Kilo	No.	Date Modified	Col5	Col6	Col7
0	.	0 B	0	07/12/2004	.	07/12/2004	0 B
1	..	0 B	1	07/12/2004	..	07/12/2004	0 B
2	admin.jar	1 KB	2	05/11/2004	admin.jar	05/11/2004	1 KB
3	applib	0 B	3	07/12/2004	applib	07/12/2004	0 B
4	applications	0 B	4	07/12/2004	applications	07/12/2004	0 B
5	config	0 B	5	07/12/2004	config	07/12/2004	0 B
6	connectors	0 B	6	07/12/2004	connectors	07/12/2004	0 B
7	database	0 B	7	07/12/2004	database	07/12/2004	0 B
8	default-web-app	0 B	8	07/12/2004	default-web-app	07/12/2004	0 B
9	iiop.jar	1,290 KB	9	05/11/2004	iiop.jar	05/11/2004	1,290 KB
10	iiop_gen_bin.jar	37 KB	10	05/11/2004	iiop_gen_bin.jar	05/11/2004	37 KB
11	iiop_rmic.jar	144 KB	11	05/11/2004	iiop_rmic.jar	05/11/2004	144 KB
12	jazn	0 B	12	07/12/2004	jazn	07/12/2004	0 B
13	jazn.jar	266 KB	13	05/11/2004	jazn.jar	05/11/2004	266 KB
14	jazncore.jar	553 KB	14	05/11/2004	jazncore.jar	05/11/2004	553 KB
15	jaznplugin.jar	12 KB	15	05/11/2004	jaznplugin.jar	05/11/2004	12 KB
16	jsp	0 B	16	07/12/2004	jsp	07/12/2004	0 B
17	lib	0 B	17	07/12/2004	lib	07/12/2004	0 B
18	loadbalancer.jar	1 KB	18	05/11/2004	loadbalancer.jar	05/11/2004	1 KB
19	log	0 B	19	07/12/2004	log	07/12/2004	0 B
20	oc4j.jar	5,696 KB	20	05/11/2004	oc4j.jar	05/11/2004	5,696 KB
21	oc4jclient.jar	1,202 KB	21	05/11/2004	oc4jclient.jar	05/11/2004	1,202 KB
22	oc4j_interop.jar	4 KB	22	05/11/2004	oc4j_interop.jar	05/11/2004	4 KB
23	ojspc.jar	1 KB	23	05/11/2004	ojspc.jar	05/11/2004	1 KB

[Figure 9-2](#) shows the same table, but nested inside a `panelGroupLayout` component, which cannot stretch its child components (for clarity, a dotted red outline has been

placed around the `panelGroupLayout` component). The table component displays only a certain number of columns and rows, determined by properties on the table.

Figure 9-2 Table Inside a Component That Does Not Stretch Its Child Components



Performance Tip:

The cost of geometry management is directly related to the complexity of child components. Therefore, try minimizing the number of child components that are under a parent geometry-managed component.

Nesting Components Inside Components That Allow Stretching

Even though you choose a component that can stretch its child components, only the following components will actually stretch:

- `decorativeBox` (when configured to stretch)
- `deck`
- `inputText` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox` (when configured to stretch)
- `panelCollection`
- `panelDashboard` (when configured to stretch)
- `panelGridLayout` (when configured to stretch)

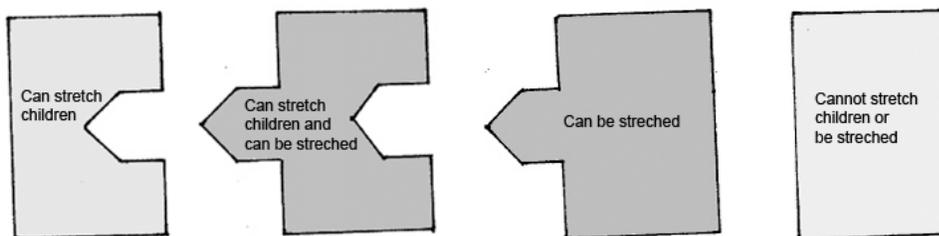
- `panelGroupLayout` (with the `layout` attribute set to `scroll` or `vertical`)
- `panelHeader` (when configured to stretch)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `showDetailHeader` (when configured to stretch)
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following layout components cannot be stretched when placed inside a facet of a component that stretches its child components:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (with the `layout` attribute set to `default` or `horizontal`)
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `tableLayout` (MyFaces Trinidad component)

One interesting way to think about geometry management and resizing is to think of components as being one of four types of puzzle pieces, as shown in [Figure 9-3](#).

Figure 9-3 Four Categories of Components for Geometry Management

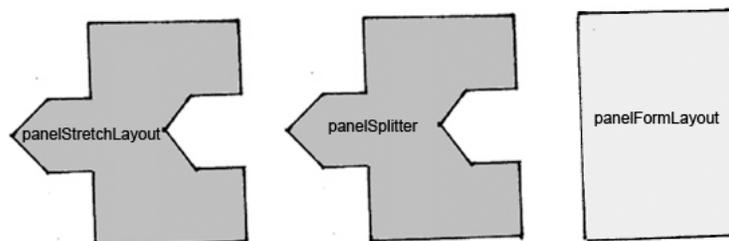


You can only place components that can be stretched inside components that stretch their children. If you want to use a component that does not stretch, within the facet of component that stretches its child components, you must wrap it in a **transition** component. Transition components can be stretched but do not stretch their children. Transition components must always be used between a component that stretches its children and a component that does not stretch. If you do not, you may see unexpected results when the component renders.

For example, suppose you want to have a form appear in one side of a `panelSplitter` component. Say your root component is the `panelStretchLayout`, and so is the first component on your page. You add a `panelSplitter` component (configured to default

settings) as a child to the `panelStretchLayout` component, and to the first facet of that component, you add a `panelFormLayout` component. [Figure 9-4](#) shows how those components would fit together. Notice that the `panelFormLayout` component cannot "fit" into the `panelSplitter` component because the `panelSplitter` can stretch its children and so will attempt to stretch the `panelFormLayout`, but the `panelFormLayout` cannot be stretched.

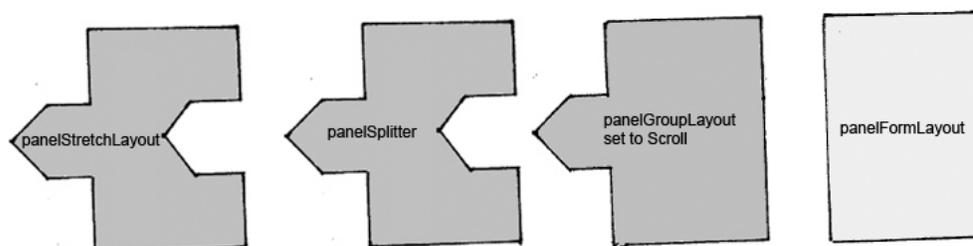
Figure 9-4 Order of Components in One Layout Scenario



When a component does not "fit" into a component that stretches children, you may get unexpected results when the browser attempts to render the component.

To have a valid layout, when you want to use a component that does not stretch in a component that stretches its children, you must use a transition component. To fix the `panelFormLayout` example, you could surround the `panelFormLayout` component with a `panelGroupLayout` component set to `scroll`. This component stretches, but does not stretch its children, as shown in [Figure 9-5](#).

Figure 9-5 Order of Components in Second Layout Scenario



In this case, all the components fit together. The `panelGroupLayout` component will not attempt to stretch the `panelFormLayout`, and so it will correctly render. And because the `panelGroupLayout` component can be stretched, the layout will not break between the components that can and cannot stretch.

 **Tip:**

Do not attempt to stretch any of the components in the list of components that cannot stretch by setting their width to 100%. You may get unexpected results. Instead, surround the component to be stretched with a component that can be stretched.

The `panelGroupLayout` component set to `scroll` is a good container for components that cannot stretch, when you want to use those components in layout with components that do stretch.

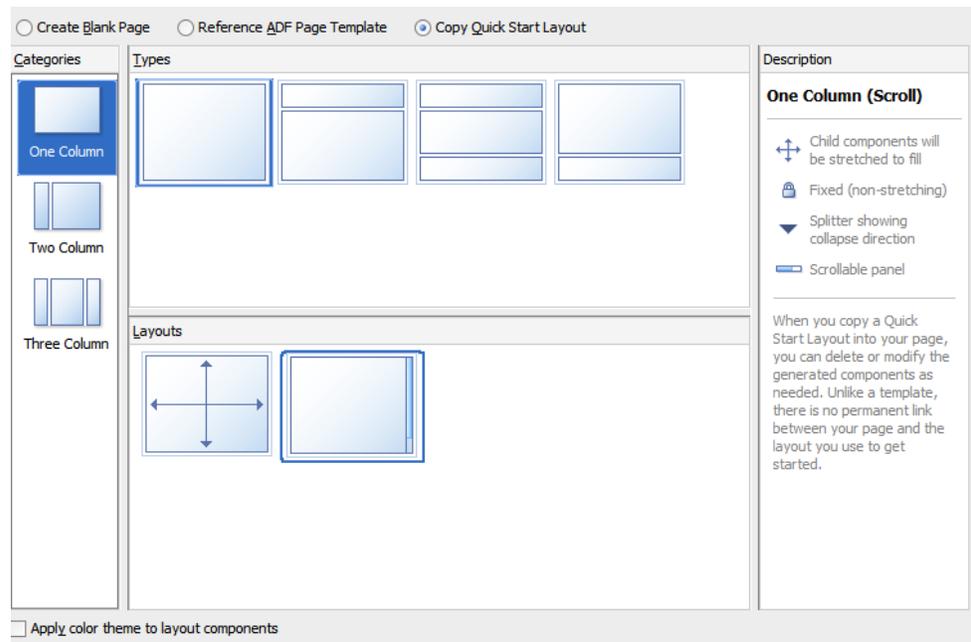
 **Tip:**

If you know that you always want your components to stretch or not to stretch based on the parent's settings, then consider setting the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter to `auto`. See [Geometry Management for Layout and Table Components](#).

Using Quick Start Layouts

When you use the New Gallery Wizard to create a JSF page (or a page fragment), you can choose from a variety of predefined quick start layouts. When you choose one of these layouts, JDeveloper adds the necessary components and sets their attributes to achieve the look and behavior you want. In addition to saving time, when you use the quick layouts, you can be sure that layout components are used together correctly to achieve the desired geometry management.

You can choose from one-, two-, and three-column formats. Within those formats, you can choose how many separate panes will be displayed in each column, and if those panes can stretch or remain a fixed size. [Figure 9-6](#) shows the different layouts available in the two-column format.

Figure 9-6 Quick Layouts

Along with adding layout components, you can also choose to apply a theme to the chosen quick layout. These themes add color styling to some of the components used in the quick start layout. To see the color and where it is added, see [Quick Start Layout Themes](#). For information about themes, see [Customizing the Appearance Using Styles and Skins](#)

For information about creating pages using the quick layouts, see [Creating a View Page](#).

Tips for Using Geometry-Managed Components

To ensure your page is displayed as expected in all browsers, use one of the quick layouts provided by JDeveloper when you create a page. These layouts ensure that the correct components are used and configured properly. See [Using Quick Start Layouts](#).

Best Practice:

Use quick start layouts to avoid layout display issues.

However, if you wish to create your layout yourself, follow these tips for creating a layout that includes both stretched and flowing components:

- Place the page contents inside a root component that performs geometry management, either `panelStretchLayout`, `panelGridLayout` with `gridRow` and `gridCell` components, `panelSplitter`, `panelAccordion` with a `showDetailItem`, or `panelTabbed` with a `showDetailItem`.

- Never specify a height value with percent units. Instead, build a component structure out of components that support being stretched and that stretch their child components. See [Nesting Components Inside Components That Allow Stretching](#).
- Inside this stretchable structure, create islands of nonstretched or flowing components by using transition components, such as the `panelGroupLayout` component with the `layout` attribute set to `scroll`. This component will provide the transition between stretched and flowing components because it supports being stretched but will not stretch its child components.
- Never try to stretch something vertically inside a nonstretched or flowing container because it will not act consistently across web browsers.
- For components contained in a parent flowing component (that is, a component that does not stretch its children), do not set widths greater than 95%. If you do, you may get unexpected results.
 - If the parent component is 768 pixels or greater, set the `styleClass` attribute on the component to be stretched to `AFStretchWidth`. This style will stretch the component to what appears to be 100% of the parent container, taking into account different browsers and any padding or borders on the parent.
 - If the parent component is 768 pixels or less, set the `styleClass` attribute on the component to be stretched to `AFAuxiliaryStretchWidth`. This style will stretch the component to what appears to be 100% of the parent container, taking into account different browsers and any padding or borders on the parent.

 **Note:**

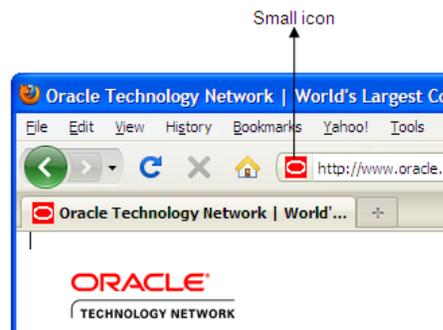
The two different styles are needed due to how Microsoft Internet Explorer 7 computes widths inside scrolling containers (this has been resolved in Internet Explorer 8). Unless you can control the version of browser used to access your application, you should use these styles as described.

- Never use the `position` style.
- Ensure that the `maximized` attribute on the `document` tag is set to `true` (this is the default). For information about setting the attribute, see [How to Configure the document Tag](#).

The remainder of this chapter describes the ADF Faces layout components and how they can be used to design a page. You can find information about how each component handles stretching in the respective "What You May Need to Know About Geometry Management" sections.

How to Configure the document Tag

The `document` tag contains a number of attributes that you can configure to control behavior for the page. For example, you can configure the icon that the browser may insert into the address bar (commonly known as a favicon). [Figure 9-7](#) shows the Oracle icon in the address bar of the Firefox browser.

Figure 9-7 Small Icon Configured on the document Tag

You can also configure the tag for the following functionality:

- **Focus:** You can set which component should have focus when the page is first rendered.
- **Uncommitted data:** You can have a warning message display if a user attempts to navigate off the page and the data has not been submitted.
- **State saving:** You can override the settings in the `web.xml` file for an individual page, so that the state of the page should be saved on the client or on the server.

To configure the document tag:

1. In the Structure window, select the **af:document** node.
2. In the Properties window, expand the Common section and set the following:
 - **InitialFocusId:** Use the dropdown menu to choose **Edit**. In the Edit Property dialog, select the component that should have focus when the page first renders.
Because this focus happens on the client, the component you select must have a corresponding client component. See [Instantiating Client-Side Components](#).
 - **Maximized:** Set to `true` if you want the root component to expand to fit all available browser space. When the document tag's `maximized` attribute is set to `true`, the framework searches for a single visual root component, and stretches that component to consume the browser's viewable area, provided that the component can be stretched. Examples of components that support this are `panelStretchLayout` and `panelSplitter`. The document tag's `maximized` attribute is set to `true` by default. See [Geometry Management and Component Stretching](#).
 - **Title:** Enter the text that should be displayed in the title bar of the browser.
3. Expand the Appearance section and set the following: and for the attribute,.
 - **FailedConnectionText:** Enter the text you want to be displayed if a connection cannot be made to the server.
 - **Small Icon Source:** Enter the URI to an icon (typically 16 pixels by 16 pixels) that the browser may insert into the address bar (commonly known as a favicon). If no value is specified, each browser may do or display something different.

You can enter a space-delimited list of icons and a browser will typically display the first value it supports. For example, Microsoft Internet Explorer only supports `.ico` for favicons. So given the following value:

```
/images/small-icon.png /small-icon.ico
```

Internet Explorer will display `small-icon.ico`, while Firefox would display `small-icon.png`.

Use one forward slash (`/`) in the address if the file is located inside of the web application's root folder. Use two forward slashes (`//`) if the file located in the server's root folder.

- **Large Icon Source:** Enter the URI to an icon (typically 129 pixels by 129 pixels) that a browser may use when bookmarking a page to a device's home page, as shown in [Figure 9-8](#).

Figure 9-8 Mobile Device Displaying Large Icon



If no value is specified, each browser may do or display something different.

You can enter a space-delimited list of icons and a browser will typically display the first value it supports.

Use one forward slash (`/`) in the address if the file is located inside of the web application's root folder. Use two forward slashes (`//`) if the file located in the server's root folder.

Tip:

Different versions of the iPhone and iPad use different sized images. You can use the largest size (129 pixels by 129 pixels) and the image will be scaled to the needed size.

4. Expand the Behavior section and set **UncommittedDataWarning** to `on` if you want a warning message displayed to the user when the application detects that data has not been committed. This can happen because either the user attempts to leave the page without committing data or there is uncommitted data on the server. By default, this is set to `off`

 **Note:**

If your application does not use ADF Controller, the data is considered to be committed when it is posted to the middle tier. For example, when a user clicks a button, no warning will be displayed when navigation occurs in the middle tier regardless of whether the data was actually written to the back end.

5. Expand the Advanced section and set **StateSaving** to the type of state saving you want to use for a page.

For ADF Faces applications, you should configure the application to use client state saving with tokens, which saves page state to the session and persists a token to the client. This setting affects the application globally, such that all pages have state saved to the session and persist tokens with information regarding state.

You can override the global setting in `web.xml` to one of the following for the page:

- **client:** The state is saved fully to the client, without the use of tokens. This setting keeps the session expired messages from being displayed.
- **default:** The state of the page is based on whatever is set in `web.xml`.
- **server:** The state of the page is saved on the server.

For information about state saving, see [Configuration in web.xml](#) .

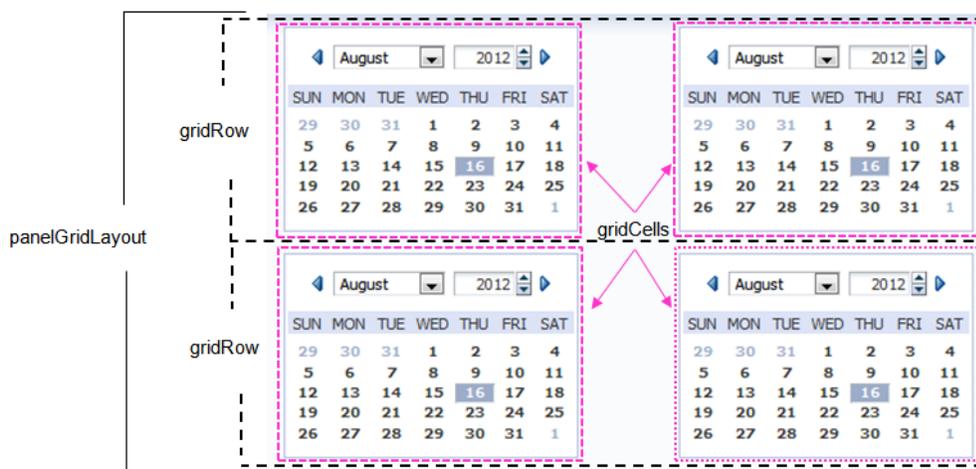
Arranging Content in a Grid

You can use the ADF Faces `panelGridLayout` component for simple structures, which gives you full control on individual cell. You can use this component when you do not need fine control over alignment. Except few cases, you can also nest a `panelGridLayout` components

Use the `panelGridLayout` component to arrange content in a grid area on a page (similar to an HTML table) and when you want the content to be able to stretch when the browser is resized. The `panelGridLayout` component provides the most flexibility of the layout components, while producing a fairly small amount of HTML elements. With it, you have full control over how each individual cell is aligned within its boundaries.

The `panelGridLayout` component uses child `gridRow` components to create rows, and then within those rows, `gridCell` components that form columns. You place components in the `gridCell` components to display your data, images, or other content.

[Figure 9-9](#) shows a `panelGridLayout` component that contains two `gridRow` components. Each of the `gridRow` components contain two `gridCell` components. Each of the `gridCell` components contain one `chooseDate` component.

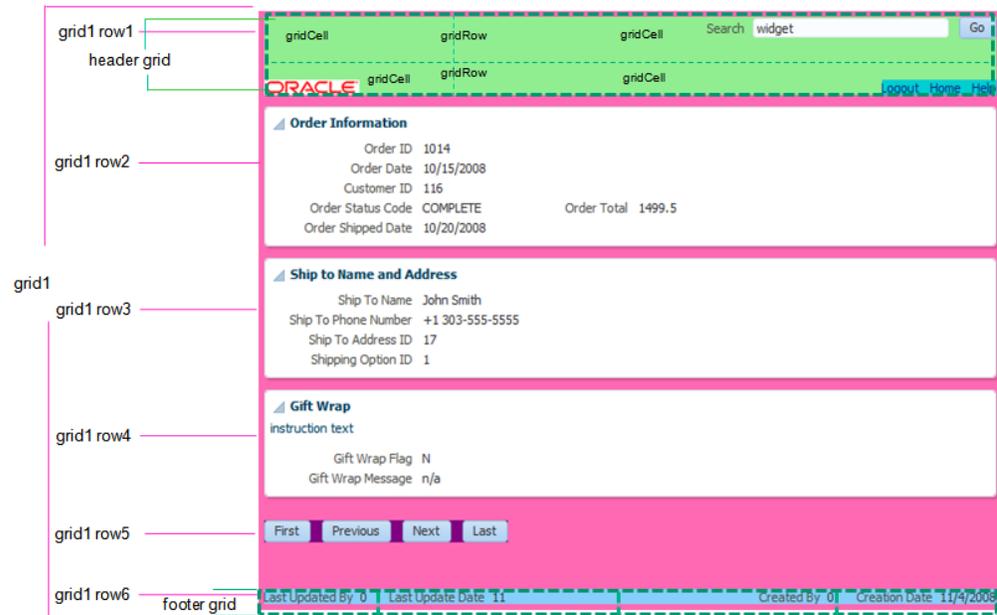
Figure 9-9 Simple Grid Layout with Two Rows Each with Two Cells

You can nest `panelGridLayout` components. In some cases, it is not safe to nest `panelGridLayout`. For example, if you set up parent `panelGridLayout` such that a cell automatically determines its size from the dimensions of the cell contents and additionally set up the contents to use a percentage size of the cell (a circular dependency), then this structure might not render properly in some browsers. Also if you set dimensions of both the parent and one or more nested grids to automatic, then the overall layout structure would not get stabilized, because the nested grid impacts the size of the parent `panelGridLayout`.

If you want to have the grid stretch its contents to fill up all available browser space, the following must be true:

- There is only one component inside of the `gridCell`
- The cell's `halign` and `valign` attributes are set to `stretch`
- The effective width and effective height of the cell are not set to be automatically determined by other cells or rows, as that would result in a circular dependency.

Each cell will then attempt to anchor the child component to all sides of the cell. If it can't (for example if the child component cannot be stretched), then the child component will be placed at the start and top of the cell. [Figure 9-10](#) shows a more complicated layout created with a parent `panelGridLayout` component (whose background is set to pink).

Figure 9-10 Complex Grid Layout Created with Nested panelGridLayout Components

The first `gridRow` component of this `panelGridLayout` contains one `gridCell` component. This `gridCell` component contains another `panelGridLayout` component for the header. This header grid contains two `gridRow` components, each with two `gridCell` components. The top right `gridCell` contains the components for search functionality, while the bottom left `gridCell` contains the Oracle logo.

The next four `gridRows` of the parent `panelGridLayout` component contain just one `gridCell` component each that holds form components and buttons. The last `gridRow` component contains one `gridCell` component that holds another `panelGridLayout` component for the footer. This footer is made up of one `gridRow` component with four `gridCell` components, each holding an `inputText` component.

When placed in a component that stretches its children, by default, the `panelGridLayout` stretches to fill its parent container. However, whether or not the content within the grid is stretched to fill the space is determined by the `gridRow` and `gridCell` components.

By default, the child contents are not stretched. The `gridRow` component determines the height. By default, the height is determined by the height of the tallest child component in the row's cells. The `gridCell` component determines the width. By default, the width of a cell is determined by the width of other cells in the column. Therefore, you must set at least one cell in a column to a determined width. You can set it to determine the width based on the component in the cell, to a fixed CSS length, or to a percentage of the remaining space in the grid.

How to Use the `panelGridLayout`, `gridRow`, and `gridCell` Components to Create a Grid-Based Layout

JDeveloper provides a dialog that declaratively creates a grid based on your input. You create a grid manually by placing a certain number of `gridRow` components into a `panelGridLayout` component. You then add `gridCell` components into the `gridRow` components, and place components that contain the actual content in the `gridCell` components. If you want to nest `panelGridLayout` components, you place the child `panelGridLayout` component into a `gridCell` component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Arranging Content in a Grid](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelGridLayout`, `gridRow`, and `gridCell` components:

1. In the Components window, from the Layout panel, drag and drop a **Panel Grid Layout** onto the JSF page.
2. In the Create Panel Grid Layout dialog, enter the number of columns and rows for the grid, set the inner and outer grid margins, then click **Next**.

When setting the inner and outer grid margins, note the following:

- **Inner Grid Margins:** Set to a fixed CSS size, for example, 2px.
 - **Columns:** Sets the value of the `marginStart` property on all `gridCell` components, except for the first one (which is handled by the **Outer Grid Margin** setting).
 - **Rows:** Sets the value of the `marginTop` property on all `gridRow` components, except for the first one (which is handled by the **Outer Grid Margin** setting).
- **Outer Grid Margins:** Set to a fixed CSS size, for example, 2px.
 - **Top:** Sets the `marginTop` property on just the top `gridRow` component.
 - **Bottom:** Sets the `marginBottom` property on just the last `gridRow` component.
 - **Left:** Sets the `marginStart` property on just the first `gridCell` component.
 - **Right:** Sets the `marginEnd` property on just the last `gridCell` component.

 **Note:**

For `marginBottom` and `marginTop`, conflicting unit types will be ignored. For example, if RowA has `marginTop` set to `2px` and RowB has `marginTop` set to `5em`, the margin will be `2px`, as that is the first unit type encountered.

When you use the Create Panel Grid Layout dialog, the `marginTop` and `marginBottom` properties are set for you and avoid this conflict.

 **Note:**

If you want the `panelGridLayout` component to stretch its children, then set the row heights to a value other than `auto` and set the cell widths to a value other than `auto`. You then need to use the Properties window to set other properties to allow stretching. See Step 5.

3. On the second page of the dialog, set the width of each cell and height of each row.
 - **Grid Width:** Sets the `width` property on each of the `gridCell` component. Set each column to one of the following:
 - `dontCare`: The width of the cell is determined by other cells in the column. This is the default.
 - `auto`: The width of the cell is determined by the components in the corresponding column. The browser first draws all those components and the width is adjusted accordingly.
 - A percentage: If you want the width of the cell's corresponding column to be a normalized percentage of the remaining space not already used by other columns, then enter a percentage, for example, `25%`.
 - A fixed CSS size: If you want to constrain the width to a fixed width, enter a fixed CSS size, for example `20px` or `20em`.

 **Note:**

Note the following:

- If you want a cell to span columns, then `width` must be set to `dontCare`.
- If cells in a column have different values for their width (for example, if one is set to `auto` and another is set to a fixed width), then the width of the column will be the largest value of the first unit type encountered.
- If all cells in a column are set to `dontCare`, then the widest cell based on its child component will determine the width of the column (as if the cells were all set to `auto`).

- **Grid Height:** Sets the `height` property on each of the `gridRow` components. Set each row to one of the following:
 - `auto`: The height of a row is determined by the components in the row. The browser first draws the child components and the height of the row is adjusted accordingly. This is the default.
 - A percentage: If the `panelGridLayout` component itself has a fixed height, or if it is being stretched by its parent component, then enter a percentage, for example `25%`. The height of the row will then be a normalized percentage of the remaining space not already used by other rows.
 - A fixed CSS length: If you want to constrain the height to a fixed height, enter a fixed CSS length, for example `10px` or `20em`.

Click **Finish**.

4. By default, the `panelGridLayout` component stretches to fill available browser space. If instead, you want to use the `panelGridLayout` component as a child to a component that does not stretch its children, then you need to change how the `panelGridLayout` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` `web.xml` parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

In the Properties window, set **DimensionsFrom** to one of the following:

- `children`: the `panelGridLayout` component will get its dimensions from its child components.

 **Note:**

If you use this setting, you cannot set the height of the child row components as percentages, because space in the `panelGridLayout` is not divided up based on availability. You can use the Properties window to change the height of the rows that you set when you completed the dialog.

- **parent**: the size of the `panelGridLayout` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.

 **Note:**

If you use this setting, you can set the height of the child row components as percentages.

- **auto**: If the parent component to the `panelGridLayout` component allows stretching of its child, then the `panelGridLayout` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelGridLayout` component will be based on the size of its child component. This is the default.
5. If you want the `panelGridLayout` to stretch its children, then you need to set the following:
 - Set **height** on the rows to a value other than `auto`.
 - Set **width** on the cells to a value other than `auto`.
 - Set **halign** on the `gridCell` components to `stretch`.
 - Set **valign** on the `gridCell` components to `stretch`.
 - Place only one child component into the `gridCell` components.
 6. If you want the cell to take up more than one column, set **ColumnSpan** to the number of columns it should span. The default is 1.

 **Note:**

If you set `columnSpan` to more than 1, then the value of the `width` attribute must be set to `dontCare`.

7. If you want the cell to take up more than one row, set **RowSpan** to the number of rows it should span. The default is 1.
8. Set **Halign** to determine the horizontal alignment for the cell's contents. If you want the contents aligned to the start of the cell (the left in LTR locale), set it to `start` (the default). You can also set it to `center` or `end`. If you want the `panelGridLayout` to stretch, then set **Halign** to `stretch` (for information about getting the `panelGridLayout` component to stretch, see Step 5.)
9. Set **Valign** to determine the vertical alignment for the cell's contents. If you want the contents aligned to the top of the cell, set it to `top` (the default). You can also set it to `middle` or `bottom`. If you want the `panelGridLayout` to stretch, then set **Valign** to `stretch` (for information about getting the `panelGridLayout` component to stretch, see Step 5).

What You May Need to Know About Geometry Management and the `panelGridLayout` Component

The `panelGridLayout` component can stretch its child components and it can also be stretched. The following components can be stretched inside the `panelGridLayout` component:

- `decorativeBox` (when configured to stretch)
- `deck`
- `calendar`
- `inputText` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox` (when configured to stretch)
- `panelCollection`
- `panelDashboard` (when configured to stretch)
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelHeader` (when configured to stretch)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `showDetailHeader` (when configured to stretch)
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside the `panelGridLayout` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch into a component that stretches its child components. Therefore, if you need to place a component that cannot be

stretched into a `gridCell` of a `panelGridLayout` component, then you must configure the `panelGridLayout`, `gridRow`, and `gridCell` components so that they do not stretch their children.

What You May Need to Know About Determining the Structure of Your Grid

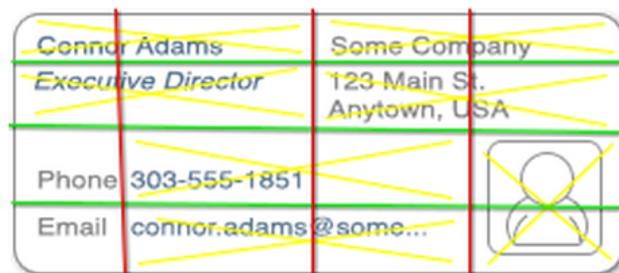
When you are given a mock-up of a page, you may not know how to break it down into a grid. Follow these tips to help determine your columns, rows, and grid separations for consecutive grids.

To design your grid:

1. Either print out the design on a piece of paper or open it up in a graphics program where you will be able to draw colored lines on top of the design.
2. Draw vertical lines representing potential column divisions in one color (for example, in red).
3. Draw horizontal lines for potential row divisions in another color, (for example, in green).
4. Now that you have a basic grid structure, use a third color (for example, yellow) to draw X marks where you see cells that need to span multiple columns or rows.

Figure 9-11 shows a design that might be broken down into four columns, four rows, and multiple places where column spans are needed.

Figure 9-11 Lines Show Potential Columns, Rows, and Spans



5. After your first attempt, you may find that your column lines really don't make sense. For example, in Figure 9-11, the two middle columns contained cells that needed to span into a nearby column. This is an indication that there should instead be two separate grids.

Use a fourth color (for example, magenta) to draw a line where the division makes sense and repeat the process again.

Figure 9-12 shows the same design but using two consecutive grids, one on top of the other.

Figure 9-12 Consecutive Grids to Simplify Column Spanning

6. Now that you can visually see where the content goes and where you need to use span columns or rows, you can code your `gridRow` and `gridCell` components. You can also accurately specify the sizes for your cells, as well as the horizontal and vertical alignments of your cells.

 **Tip:**

When you see in your grid that fields with labels span columns, instead of using the built-in labels, configure the fields to hide those labels (usually using the `simple` attribute), and instead use a separate `outputLabel` component for the label.

For example, in [Figure 9-12](#), the labels for the Phone and Email field are in the first column, while the fields themselves are in the second column. To create this layout, instead of using the labels in the corresponding `inputText` component, you would:

- a. Place the `inputText` components for the phone and email fields in the second column.
- b. Set the `simple` attribute on those components to `true`, so the built-in labels don't display.
- c. Add new `outputLabel` components to the first column for the labels.

[Figure 9-13](#) shows the final grid design. There are two vertically stacked grids. The top grid contains two columns and two rows. The bottom grid contains three columns and two rows, with the last column spanning the two rows.

Figure 9-13 Final Grid Design



 **Best Practice Tip:**

You should not have more than three layers of `panelGridLayout` components.

What You May Need to Know About Determining Which Layout Component to Use

The `panelGridLayout` component provides the most flexibility of the layout components, while producing a fairly small amount of HTML elements. With it, you have full control over how each individual cell is aligned within its boundaries. Conversely, the `panelGroupLayout` provides very little control over how individual children of the structure are presented, and the `panelStretchLayout` only produces a small number of grid structures, often requiring the nesting of multiple `panelStretchLayout` components. Nesting multiple components means more HTML elements are needed, and also that the code will be more difficult to maintain. Therefore, for complex layouts, use the `panelGridLayout` component.

Use the `panelGroupLayout` for simple structures where you don't need fine control over alignment, for example to align a series of button components. If you find yourself nesting multiple `panelGroupLayout` components, this is an indication that `panelGridLayout` would be more appropriate.

Displaying Contents in a Dynamic Grid Using a masonryLayout Component

You can use the ADF Faces masonryLayout component when you want the content to be rendered dynamically. This component takes any ADF Faces component as a child.

The masonryLayout component displays its contents in a grid that has dynamic rendering capabilities. It can take any ADF Faces component as a child, respectively called a tile. Tiles can span columns and rows. When the UI is provided, users can insert, delete, reorder and resize the tiles.

When the masonryLayout component renders, each tile is processed in the order that it occurs in the code and is positioned in the first location that accommodates it. The location is determined using the reading direction of the client browser (left-to-right or right-to-left), and then top-to-bottom. If the next available location is not large enough for the tile, a gap will be left and the tile placed in the next available space. A subsequent tile that fits may be placed in the gap. If no tiles fit, then the gap remains. When the window size changes, if necessary, the masonryLayout renders a different numbers of rows or columns, based on the size of the tiles. The size of the tiles does not change.

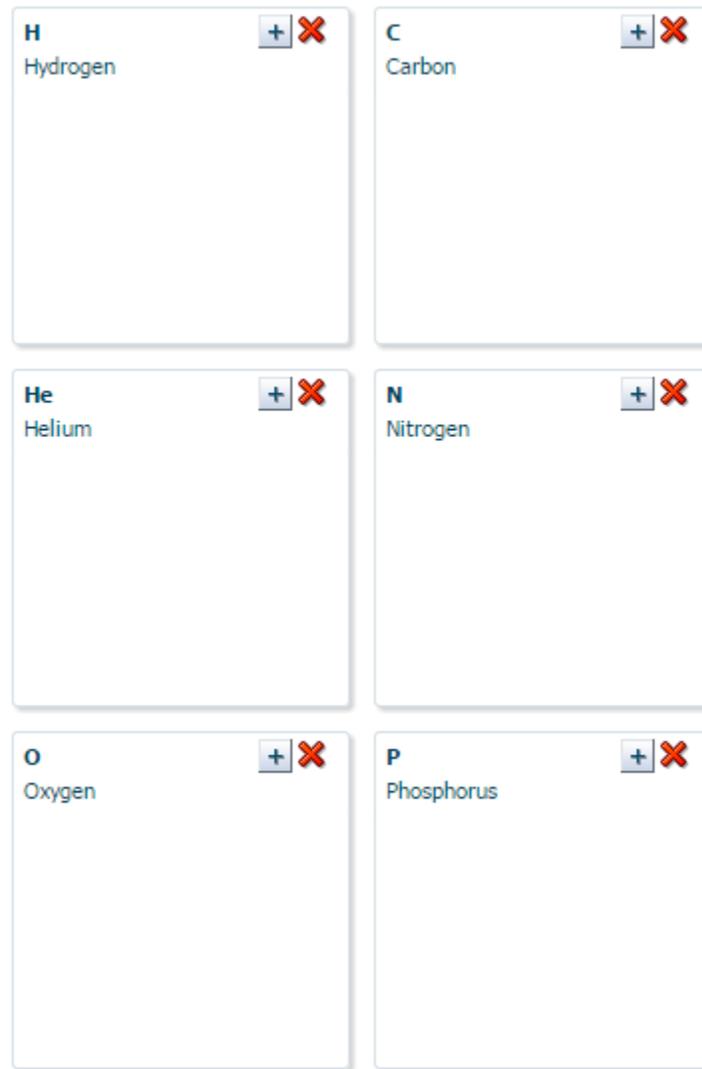
For example, [Figure 9-14](#) shows a masonryLayout component with three columns and two rows.

Figure 9-14 masonryLayout Displaying Three Columns



As the display area width is reduced, the masonryLayout reduces the number of columns to two, and the number of rows to three, while increasing the number of rows to three. The tiles remain the same size, as shown in [Figure 9-15](#).

Figure 9-15 masonryLayout Displaying the Same Tiles in a Smaller Space



If you want a set number of columns, you set a fixed width or maximum width on the layout. If you want to limit the height of the layout, you set a fixed height or maximum height on the layout and enable scrolling to handle any overflow.

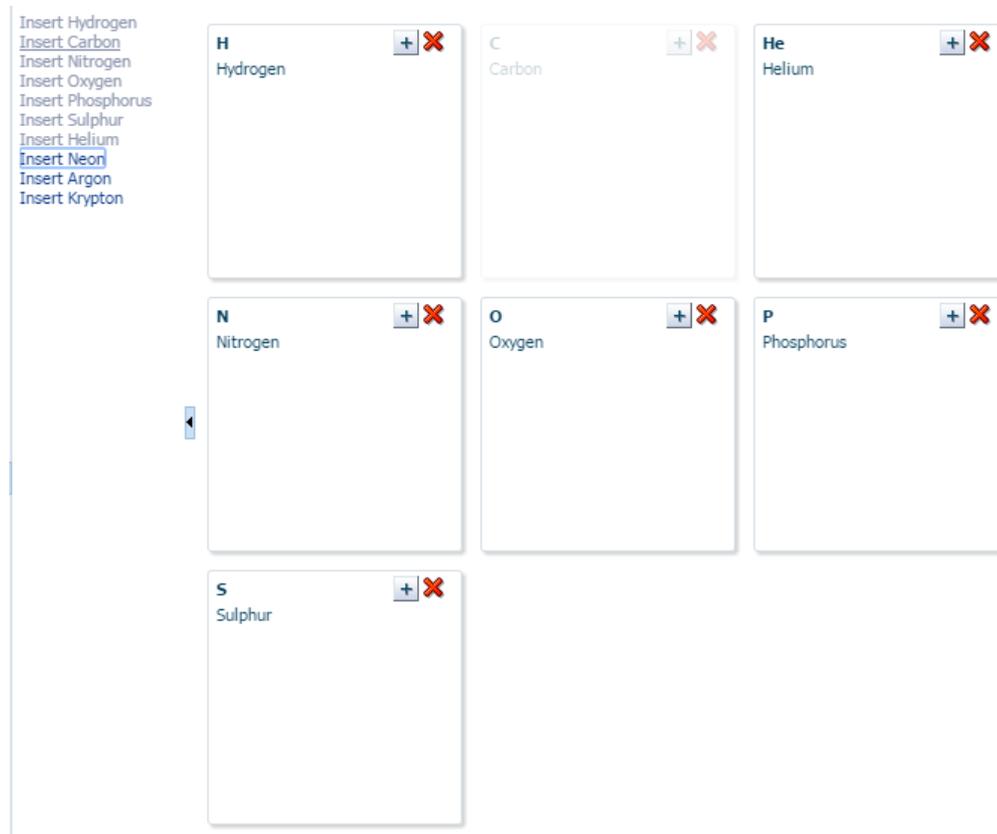
You set the size of a tile using the `AFMasonryTileSize` style classes on the tile component. For example, if you use a `panelBox` as a tile, and you want it to span two columns and 1 row, you would set the style class on the `panelBox` to `AFMasonryTileSize2x1` (all available style classes are noted in [How to Use a masonryLayout Component](#)).

Listeners are available on the `masonryLayout` component to handle resizing, reordering, inserting and deleting tiles. You need to create the code to handle these actions, as well as the UI to initiate the actions. The `masonryLayout` component doesn't fire the events - the components that render the tiles do. You need to add these components to the page and have the `masonryLayout` component listen for their events.

Instead of creating those components and wiring them to the layout, you can use the `masonryLayoutBehavior` tag. This tag provides a declarative way to use a command component to initiate the layout changes. It also renders visual changes to the layout before the component tree is actually modified. Because this opening up of space happens before the action event is sent to the server, the user will see immediate feedback while the listener for the command component modifies the component tree.

For example, [Figure 9-16](#) shows a `masonryLayout` component used in the right panel of a `panelSplitter` component. In the left panel, list items displayed as links represent each `panelBox` tile in the `masonryLayout`. When all tiles are displayed, the links are all inactive. However, if a user deletes one of the tiles, the corresponding link becomes active. The user can click the link to reinsert the tile. By using the `masonryLayoutBehavior` tag with the `commandLink` component, the user sees the movement of the tiles and the space for the inserted tile much sooner.

Figure 9-16 Links to Add Tiles Use the `masonryLayoutBehavior` tag



You do not have to use this tag to provide any insert, delete, resize, or reordering. The tag simply provides visual feedback more quickly. Without it, users would not see the visual changes until the new content is retrieved from the server.

The `masonryLayout` component will stretch if the parent component allows stretching of its child. If the parent does not stretch its children then the size of the `masonryLayout` component will be based on the contents of its child components.

How to Use a masonryLayout Component

After you add a `masonryLayout` to a page, if you want to allow insertion, deletion, reordering, or resizing of child components, you need to implement a method to handle each of those actions, as well as the components to initiate those actions. You then add any child components as tiles to the layout. If you want to allow rearranging the child components, you need to add a `componentDragSource` tag to the child components. You can also use the `masonryLayoutBehavior` tag to make the `masonryLayout` component appear more responsive to the layout changes.

To use the `masonryLayout` component:

1. In the Component Palette, from the ADF Faces panel drag and drop a **Masonry Layout** onto the page.
2. In the Property Inspector, expand the Other section.
3. By default, the layout will be dynamically sized to fit its container, based on the size of the children. If you want a set number of columns, set a fixed width or maximum width on the layout. If you want to limit the height of the layout, set a fixed height or maximum height on the layout and enable scrolling to handle any overflow.
4. From the Component Palette, drag and drop child components.
5. In the Property Inspector, with the child component selected, expand the Style section and set `StyleClass` to determine how many columns and rows the tiles should span. The available values that are supported by pre-defined style classes are:

- `AFMasonryTileSize1x1`: 1 column by 1 row
- `AFMasonryTileSize1x2`: 1 column by 2 rows
- `AFMasonryTileSize1x3`: 1 column by 3 rows
- `AFMasonryTileSize2x1`: 2 columns by 1 row
- `AFMasonryTileSize2x2`: 2 columns by 2 rows
- `AFMasonryTileSize2x3`: 2 columns by 3 rows
- `AFMasonryTileSize3x1`: 3 columns by 1 row
- `AFMasonryTileSize3x2`: 3 columns by 2 rows

Note: The `masonryLayout` component recognizes tile size style class names for spans up to and including 10x10. For tile sizes of span 3x3 and greater (up to 10x10 maximum), your application may define style classes for the desired span. Instead of using pre-defined style classes, you can create style classes with the naming pattern `AFMasonryTileSize<colSpan>x<rowSpan>`, where `colspan` and `rowspan` are the numbers indicating the respective spans in the layout grid. Ensure that you define the style classes in your skinning style sheet if you are defining your own style classes in addition to the pre-defined set. For example, the following defines a 4x4 span tile size:

```
AFMasonryTileSize4x4 {  
  -tr-rule-ref: selector(".AFMasonryTileSizeBase:alias");  
  width: 728px;  
}
```

```
height: 728px;
}
```

6. If you want users to be able to change the layout of the tiles, do the following:
 - a. In the Component Palette, from the Operations panel, drag and drop a **Component Drag Source** as a child to each of the child components that render the tiles.
 - b. Create a managed bean and implement a handler method for each associated layout change, such as insert, delete, and reorder. Once created, use the Property Inspector to bind the corresponding listener property on the `masonryLayout` component to the method.
 - c. The reordering event is considered a drop event, so you must use the Drag and Drop framework. The `masonryLayout` is the `dropTarget`, and will need to use a `dataFlavor` tag to restrict the drop. For information about creating a handler for a drop event, see [About Drag and Drop Functionality](#). The following page code shows the tags for drag and drop functionality. The child `panelBox` components are handled by an iterator.

```
<af:masonryLayout binding="#{editor.component}" id="m11"
  reorderListener="#{demoMasonryLayout.handleBasicReorder}">
  <af:dropTarget actions="MOVE"
    dropListener="#{demoMasonryLayout.handleDrop}">
    <af:dataFlavor flavorClass="javax.faces.component.UIComponent"
      discriminant="masonryTile"/>
  </af:dropTarget>
  <af:iterator var="row" varStatus="stat"
    value="#{demoMasonryLayout.basicData}" id="it1">
    <af:panelBox id="pb1" text="#{row.symbol}"
      showDisclosure="false"
      styleClass="#{row.symbol == 'H' ?
        'AFMasonryTileSize2x1' :
        'AFMasonryTileSize1x1'}">
      <af:componentDragSource discriminant="masonryTile"/>
      <af:outputText id="ot1" value="#{row.name}"/>
    </af:panelBox>
  </af:iterator>
</af:masonryLayout>
```

7. If you are not going to use the `masonryLayoutBehavior` tag, then add components to the page to initiate each of the layout changes. Bind their listener properties to the handler methods created in Step 6. Skip the remaining steps.
8. If you wish to use a `masonryLayoutBehavior` tag, drag and drop a command component that will be used to initiate the layout change.
9. In the Component Palette, from the ADF Faces panel, drag a **Masonry Layout Behavior** tag and drop it as a child to each command component.
10. In the Property Inspector, enter the following:
 - **for:** Enter the ID for the associated `masonryLayout` component
 - **index:** Enter an EL expression that resolves to a method that determines the index at which the component will be inserted into the layout. When you use the `masonryLayoutBehavior` tag, a placeholder element is inserted into the DOM tree where the actual component will be rendered once it is returned

from the server. Because the insertion placeholder gets added before the insertion occurs on the server, you must specify the location where you are planning to insert the child component, and this location must be the same location specified by your handler methods that do the actual change, in order to preserve the context both before and after the change.

- **operation:** Enter the layout change as one of the following:
 - insert (default)
 - delete
 - resize

The following page code shows links used with a `masonryLayoutBehavior` tag to resize and delete tiles from the layout. Code for other layout changes would be similar.

```
<af:link id="cil2" shortDesc="Expand" partialSubmit="true"
        rendered="#{!row.expanded}" icon="/images/
field_groups_add_ena.png"
        hoverIcon="/images/field_groups_add_ovr.png"
        depressedIcon="/images/field_groups_add_dwn.png">
  <af:setPropertyListener type="action" from="#{row.symbol}"
                        to="#{demoMasonryLayout.currentSymbol}"/>
  <af:masonryLayoutBehavior operation="resize" for="m11"

sizeStyleClass="#{row.expandedSizeStyleClass}"/>
</af:link>
<af:link id="cil3" shortDesc="Collapse" partialSubmit="true"
        rendered="#{row.expanded}" icon="/images/
field_groups_remove_ena.png"
        hoverIcon="/images/field_groups_remove_ovr.png"
        depressedIcon="/images/field_groups_remove_dwn.png">
  <af:setPropertyListener type="action" from="#{row.symbol}"
                        to="#{demoMasonryLayout.currentSymbol}"/>
  <af:masonryLayoutBehavior operation="resize" for="m11"
                        sizeStyleClass="#{row.sizeStyleClass}"/>
</af:link>
<af:link id="cil1" shortDesc="Delete" partialSubmit="true"
        icon="/images/delete_ena.png"
        hoverIcon="/images/delete_ovr.png"
        depressedIcon="/images/delete_dwn.png">
  <af:setPropertyListener type="action" from="#{row.symbol}"
                        to="#{demoMasonryLayout.currentSymbol}"/>
  <af:masonryLayoutBehavior for="m11" operation="delete"/>
</af:link>
```

Achieving Responsive Behavior Using `matchMediaBehavior` Tag

Using the `matchMediaBehavior` tag you can create a responsive user interface. You can control the alignment of almost every component in the screen to the available viewport size.

The `matchMediaBehavior` tag is a declarative way to define properties for a component for different `@media` rules. It uses the standard media queries and matches them with the rules specified by each of the behavior tags. Once the rule matches, it applies the property defined through the behavior tag on the component and refreshes the component to display the intended change. If the media query does not match any rules, a default value is used.

The three main attributes of the `matchMediaBehavior` tag are as follows:

- `MatchedPropertyValue`: The value that should be set in the property name when there is a media match.
- `MediaQuery`: The media query on which the tag is listening on.
- `PropertyName`: The property that has to undergo a change during a media operation.



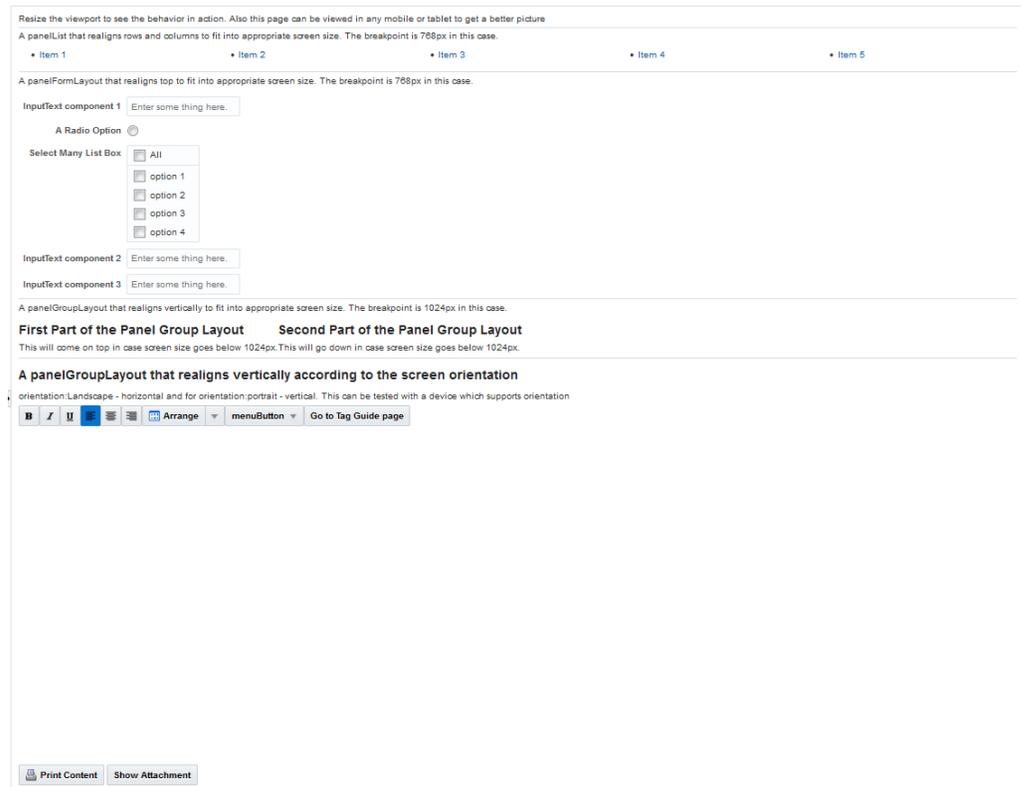
Note:

You can specify EL expressions for the `MatchedPropertyValue` and `MediaQuery` attributes. In the Expression Builder dialog, you can directly type the expression or select values from variables and operators to create an expression. See [How to Create an EL Expression](#).

The `matchMediaBehavior` tag enables you to create a responsive user interface where components align themselves to the available width and height of the device. This behavior can be added under layout components or components that are capable of realigning on Partial Page Rendering (PPR). The `matchMediaBehavior` tag gives you more control over component level attributes unlike `masonryLayout` component, which deals with overall layout. The usage of `matchMediaBehavior` tag is explained in the following example.

In the [Figure 9-17](#) example, the content is displaying horizontally across the page width.

Figure 9-17 matchMediaBehavior — Horizontal Layout



In the [Figure 9-18](#), the content is displayed vertically when the page is viewed in a smaller screen area. In this case, the components under `panelListLayout` and `panelFormLayout` realign themselves vertically when the screen area is below 786 pixels and the components under `panelGroupLayout` realign themselves vertically when the screen area is below 1024 pixels.

Figure 9-18 matchMediaBehavior — Vertical Layout

Resize the viewport to see the behavior in action. Also this page can be viewed in any mobile or tablet to get a better picture

A panelList that realigns rows and columns to fit into appropriate screen size. The breakpoint is 768px in this case.

- Item 1
- Item 2
- Item 3
- Item 4
- Item 5

A panelFormLayout that realigns top to fit into appropriate screen size. The breakpoint is 768px in this case.

InputText component 1

Enter some thing here.

A Radio Option

Select Many List Box

All

option 1

option 2

option 3

option 4

InputText component 2

Enter some thing here.

InputText component 3

Enter some thing here.

A panelGroupLayout that realigns vertically to fit into appropriate screen size. The breakpoint is 1024px in this case.

First Part of the Panel Group Layout

This will come on top in case screen size goes below 1024px.

Second Part of the Panel Group Layout

This will go down in case screen size goes below 1024px.

A panelGroupLayout that realigns vertically according to the screen orientati...

orientation:Landscape - horizontal and for orientation:portrait - vertical. This can be tested with a device which supports orientation

B I U   

 Arrange   menuButton  [Go to Tag Guide page](#)

 Print Content  Show Attachment

The following code example describes the usage of matchMediaBehavior tag.

```
<af:outputText value="A panelList that realigns rows and columns to fit into
appropriate screen size.
        The breakpoint is 768px in this case." id="ot2"/>
        <af:panelList maxColumns="5" id="p11" shortDesc="Links" rows="1"
inlineStyle="text-align:left;">
        <af:matchMediaBehavior propertyName="maxColumns"
matchedPropertyValue="2"
```

```

                    mediaQuery="screen and (max-width: 768px)"/>
<af:matchMediaBehavior propertyName="rows" matchedPropertyValue="3"
                    mediaQuery="screen and (max-width: 768px)"/>
<af:commandLink id="link_id_1">Item 1</af:commandLink>
<af:commandLink id="link_id_2">Item 2</af:commandLink>
<af:commandLink id="link_id_3">Item 3</af:commandLink>
<af:commandLink id="link_id_4">Item 4</af:commandLink>
<af:commandLink id="link_id_5">Item 5</af:commandLink>
</af:panelList>
<af:outputText value="A panelFormLayout that realigns top to fit into appropriate
screen size.
                    The breakpoint is 768px in this case." id="ot3"/>
<af:panelFormLayout id="pfl1" clientComponent="true"
labelAlignment="start">
    <af:matchMediaBehavior propertyName="labelAlignment"
matchedPropertyValue="top"
                    mediaQuery="screen and (max-width: 768px)"/>
    <af:inputText label="InputText component 1" placeholder="Enter some
thing here." id="it0"/>
    <af:selectBooleanRadio id="rb" group="rbGroup" shortDesc="shortDesc
text" label="A Radio Option"/>
    <af:selectManyListbox id="rs" label="Select Many List Box"
shortDesc="Select Option">
        <af:selectItem label="option 1" id="si5"/>
        <af:selectItem label="option 2" id="si6"/>
        <af:selectItem label="option 3" id="si7"/>
        <af:selectItem label="option 4" id="si8"/>
    </af:selectManyListbox>
    <af:inputText label="InputText component 2" placeholder="Enter some
thing here." id="it1"/>
    <af:inputText label="InputText component 3" placeholder="Enter some
thing here." id="it2"/>
    <f:facet name="footer"></f:facet>
</af:panelFormLayout>
<af:outputText value="A panelGroupLayout that realigns vertically to fit into
appropriate screen size.
                    The breakpoint is 1024px in this case." id="ot4"/>
<af:panelGroupLayout id="pg15" clientComponent="true"
layout="horizontal">
    <af:matchMediaBehavior propertyName="layout"
matchedPropertyValue="vertical"
                    mediaQuery="screen and (max-width: 1024px)"/>
    <af:panelStretchLayout id="ps11" dimensionsFrom="children">
        <f:facet name="center">
            <af:panelGroupLayout id="pg13">
                <af:panelHeader text="First Part of the Panel Group Layout"
id="ph7" headerLevel="6"></af:panelHeader>
                <af:outputText value="This will come on top in case screen
size goes below 1024px." id="ot5"/>
            </af:panelGroupLayout>

```

The following code example describes the usage of matchMediaBehavior tag for listView. The matchMediaBehavior component works with the fetchSize property of the listView component only if listView includes both fetchSize and rows attributes.

```

<af:listView var="item" emptyText="empty" fetchSize="7" rows="7" id="lv1"
value="#{mybean.mapEntries}">
    <af:matchMediaBehavior matchedPropertyValue="4" propertyName="rows"
mediaQuery="screen and (max-height: 500px)"/>
    <af:matchMediaBehavior matchedPropertyValue="4" propertyName="fetchSize"
mediaQuery="screen and (max-height: 500px)"/>

```

```

        <af:matchMediaBehavior matchedPropertyValue="7" propertyName="rows"
mediaQuery="screen and (min-height: 500px)"/>
        <af:matchMediaBehavior matchedPropertyValue="7" propertyName="fetchSize"
mediaQuery="screen and (min-height: 500px)"/>
        <af:listItem id="li1">
            <af:panelGroupLayout layout="horizontal" id="pgl2">
                <af:outputFormatted value="{item.key}" id="of1"/>
                <af:spacer width="10" height="10" id="s1"/>
                <af:outputFormatted value="{item.value}" id="of2"/>
            </af:panelGroupLayout>
        </af:listItem>
    </af:listView>

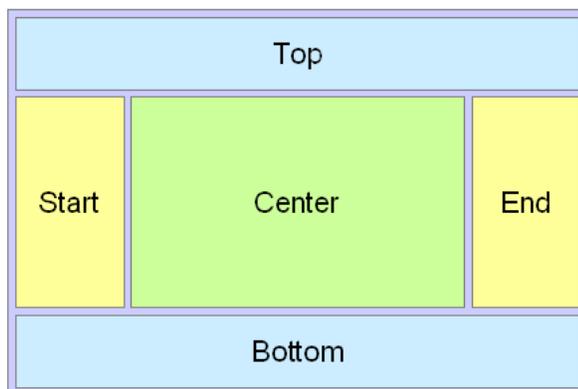
```

Arranging Contents to Stretch Across a Page

The ADF Faces `panelStretchLayout` component stretch the components placed within its facets. If you want to stretch the contents when the browser is resized, you can use this component.

Use the `panelStretchLayout` component to arrange content in defined areas on a page and when you want the content to be able to stretch when the browser is resized. The `panelStretchLayout` component is one of the components that can stretch components placed in its facets. [Figure 9-19](#) shows the component's facets: top, bottom, start, end, and center.

Figure 9-19 Facets in the `panelStretchLayout` Component



Note:

[Figure 9-19](#) shows the facets when the language reading direction of the application is configured to be left-to-right. If instead the language direction is right-to-left, the `start` and `end` facets are switched.

When you set the height of the `top` and `bottom` facets, any contained components are stretched up to fit the height. Similarly, when you set the width of the `start` and `end` facets, any components contained in those facets are stretched to that width. If no components are placed in the facets, then that facet does not render. That is, that facet will not take up any space. If you want that facet to take up the set space but remain blank, insert a `spacer` component. See [Separating Content Using Blank Space](#)

or `Lines`. Child Components components in the `center` facet are then stretched to fill up any remaining space. For information about component stretching, see [Geometry Management and Component Stretching](#).

Instead of setting the height of the top or bottom facet, or width of the start or end facet to a dimension, you can set the height or width to `auto`. This allows the facet to size itself to use exactly the space required by the child components of the facet. Space will be allocated based on what the web browser determines is the required amount of space to display the facet content.

Performance Tip:

Using `auto` as a value will degrade performance of your page. You should first attempt to set a height or width and use the `auto` attribute sparingly.

The File Explorer application uses a `panelStretchLayout` component as the root component in the template. Child components are placed only in the `center` and `bottom` facets. Therefore, whatever is in the `center` facet stretches the full width of the window, and from the top of the window to the top of the `bottom` facet, whose height is determined by the `bottomHeight` attribute. The following example shows abbreviated code from the `fileExplorerTemplate` file.

```
<af:panelStretchLayout
  bottomHeight="#{attrs.footerGlobalSize}">
  <f:facet name="center">
    <af:panelSplitter orientation="vertical" ...>
    .
    .
    .
    </af:panelSplitter
  </f:facet>
  <f:facet name="bottom">
    <af:panelGroupLayout layout="vertical">
    .
    .
    .
    </af:panelGroupLayout>
  </f:facet>
</af:panelStretchLayout>
```

The template uses an EL expression to determine the value of the `bottomHeight` attribute. This expression resolves to the value of the `footerGlobalSize` attribute defined in the template, which by default is 0. Any page that uses the template can override this value. For example, the `index.jspx` page uses this template and sets the value to 30. Therefore, when the File Explorer application renders, the contents in the `panelStretchLayout` component begin 30 pixels from the bottom of the page.

How to Use the `panelStretchLayout` Component

The `panelStretchLayout` component cannot have any direct child components. Instead, you place components within its facets. The `panelStretchLayout` is one of the components that can be configured to stretch any components in its facets to fit the browser. You can nest `panelStretchLayout` components. See [Nesting Components Inside Components That Allow Stretching](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Arranging Contents to Stretch Across a Page](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelStretchLayout` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Stretch Layout** onto the JSF page.
2. In the Properties window, expand the Common section and set the attributes as needed.

When there are child components in the `top`, `bottom`, `start`, and `end` facets, these components occupy space that is defined by the `topHeight`, `bottomHeight`, `startWidth`, and `endWidth` attributes. For example, `topHeight` attribute specifies the height of the `top` facet, and `startWidth` attribute specifies the width of the `start` facet. Child components in `top` and `bottom` facets are stretched up to the height set by `topHeight` and `bottomHeight` attributes, respectively, and child components in `start` and `end` facets are stretched up to the width set by `startWidth` and `endWidth` attributes, respectively. Instead of setting a numeric dimension, you can set the `topHeight`, `bottomHeight`, `startWidth` and `endWidth` attributes to `auto` and the browser will determine the amount of space required to display the content in the facets.

 **Note:**

If you set a facet to use `auto` as a value for the width or height of that facet, the child component does not have to be able to stretch. In fact, it must use a stable, standalone width that is not dependent upon the width of the facet.

For example, you should not use `auto` on a facet whose child component can stretch their children automatically. These components have their own built-in stretched widths by default which will then cause them to report an unstable `offsetWidth` value, which is used by the browser to determine the amount of space.

Additionally, you should not use `auto` in conjunction with a child component that uses a percentage length for its width. The facet content cannot rely on percentage widths or be any component that would naturally consume the entire width of its surrounding container.

If you do not explicitly specify a value, by default, the value for the `topHeight`, `bottomHeight`, `startWidth`, and `endWidth` attributes is 50 pixels each. The widths of the `top` and `bottom` facets, and the heights of the `start` and `end` facets are derived from the width and height of the parent component of `panelStretchLayout`.

 **Tip:**

If a facet does not contain a child component, it is not rendered and therefore does not take up any space. You must place a child component into a facet in order for that facet to occupy the configured space.

3. The `panelStretchLayout` component can be configured to stretch to fill available browser space, or if you want to place the `panelStretchLayout` component inside a component that does *not* stretch its children, you can configure the `panelStretchLayout` component to not stretch.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

Set **DimensionsFrom** to one of the following:

- `children`: Instead of stretching, the `panelStretchLayout` component will get its dimensions from its child component.

 **Note:**

If you use this setting, you cannot use a percentage to set the height of the `top` and `bottom` facets. If you do, those facets will try to get their dimensions from the size of this `panelStretchLayout` component, which will not be possible, as the `panelStretchLayout` component will be getting its height from its contents, resulting in a circular dependency. If a percentage is used for either facet, it will be disregarded and the default `50px` will be used instead.

Additionally, you cannot set the height of the `panelStretchLayout` component (for example through the `inlineStyle` or `styleClass` attributes) if you use this setting. Doing so would cause conflict between the `panelStretchLayout` height and the child component height.

- **parent:** the size of the `panelStretchLayout` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container (that is, the `panelStretchLayout` component will stretch).
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
 - **auto:** If the parent component to the `panelStretchLayout` component allows stretching of its child, then the `panelStretchLayout` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelStretchLayout` component will be based on the size of its child component.
4. To place content in the component, drag and drop the desired component into any of the facets. If you want the child component to stretch, it must be a component that supports being stretched. See [What You May Need to Know About Geometry Management and the `panelStretchLayout` Component](#).

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component. Child components must also be able to be stretched in order for all contained components to stretch.

 **Tip:**

If any facet is not visible in the visual editor:

- a. Right-click the `panelStretchLayout` component in the Structure window.
- b. From the context menu, choose **Facets - Panel Stretch Layout >facet name**. Facets in use on the page are indicated by a checkmark in front of the facet name.

What You May Need to Know About Geometry Management and the `panelStretchLayout` Component

The `panelStretchLayout` component can stretch its child components and it can also be stretched. The following components can be stretched inside the facets of the `panelStretchLayout` component:

- `decorativeBox` (when configured to stretch)
- `deck`
- `calendar`
- `inputText` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox` (when configured to stretch)
- `panelCollection`

- `panelDashboard` (when configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelHeader` (when configured to stretch)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `showDetailHeader` (when configured to stretch)
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a facet of the `panelStretchLayout` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch into facets of a component that stretches its child components. Therefore, if you need to place a component that cannot be stretched into a facet of the `panelStretchLayout` component, wrap that component in a transition component that can stretch.

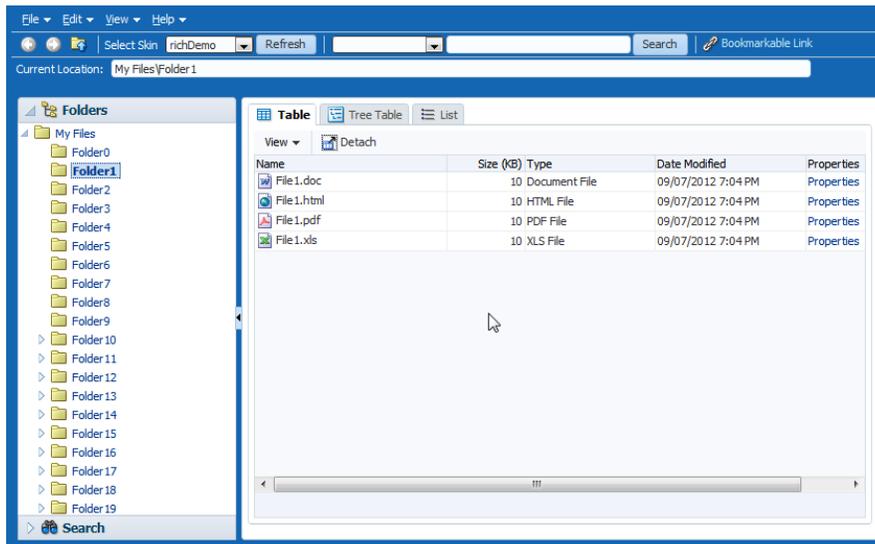
For example, if you want to place content in a `panelBox` component (configured to not stretch) within a facet of the `panelStretchLayout` component, you could place a `panelGroupLayout` component with its `layout` attribute set to `scroll` in a facet of the `panelStretchLayout` component, and then place the `panelBox` component in that `panelGroupLayout` component. See [Nesting Components Inside Components That Allow Stretching](#).

Using Splitters to Create Resizable Panes

Using the ADF Faces `panelSplitter` component, you can display the content in multiple panes. This component stretches itself and also its child components. When a panel is collapsed, the content in the other panel is automatically resized.

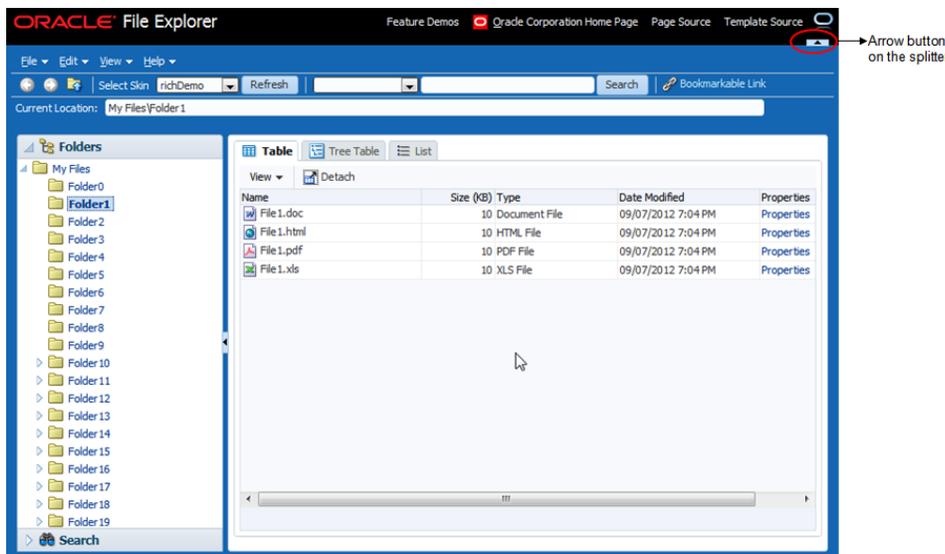
When you have groups of unique content to present to users, consider using the `panelSplitter` component to provide multiple panes separated by adjustable splitters. The ADF Faces Components Demo application uses a `panelSplitter` to separate the component demo area from the editor area, as shown in [Figure 9-20](#). Users can change the size of the panes by dragging the splitter, and can also collapse and restore the panel that displays the editor. When a panel is collapsed, the panel contents are hidden; when a panel is restored, the contents are displayed.

Figure 9-20 ADF Faces Components Demo Application Uses `panelSplitter` to Separate Contents



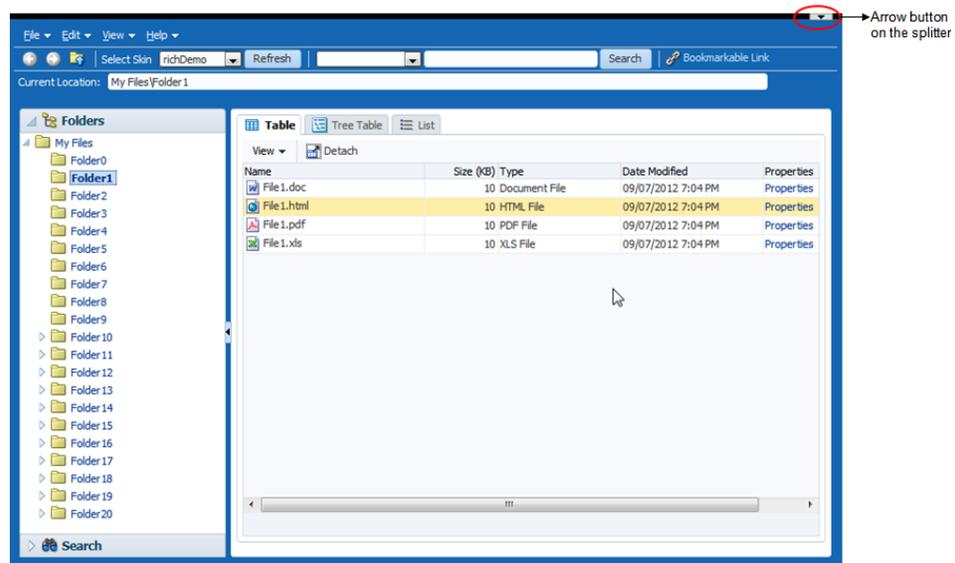
The `panelSplitter` component lets you organize contents into two panes separated by an adjustable splitter. The panes can either line up on a horizontal line (as does the splitter shown in Figure 9-20) or on a vertical line. The ADF Faces Components Demo application uses another `panelSplitter` component to separate the application's global menu from the main body of the page. Figure 9-21 shows the `panelSplitter` component expanded to show the menu, which includes access to the documentation and source.

Figure 9-21 `panelSplitter` with a Vertical Split Expanded



Clicking the arrow button on a splitter collapses the panel that holds the global menu, and the menu items are no longer shown, as shown in Figure 9-22.

Figure 9-22 panelSplitter with a Vertical Split Collapsed



You place components inside the facets of the `panelSplitter` component. The `panelSplitter` component uses geometry management to stretch its child components at runtime. This means when the user collapses one panel, the contents in the other panel are explicitly resized to fill up available space.

 **Note:**

While the user can change the values of the `splitterPosition` and `collapsed` attributes by resizing or collapsing the panes, those values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see [Allowing User Customization on JSF Pages](#).

How to Use the `panelSplitter` Component

The `panelSplitter` component lets you create two panes separated by a splitter. Each splitter component has two facets, namely, `first` and `second`, which correspond to the first panel and second panel, respectively. Child components can reside inside the facets only. To create more than two panes, you nest the `panelSplitter` components.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Splitters to Create Resizable Panes](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelSplitter` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Splitter** onto the JSF page.
2. In the Properties window, expand the Common section.
3. Set **Orientation** to `vertical` to create two vertical panes (one on top of the other). By default, the value is `horizontal`, which means horizontal panes are placed left-to-right (or right-to-left, depending on the language reading direction).
4. Set **SplitterPosition** and **PositionedFromEnd** to determine the initial placement of the splitter. By default, the value of the `splitterPosition` attribute is 200 pixels, and the `positionedFromEnd` attribute is `false`. This setting means that ADF Faces measures the initial position of the adjustable splitter from the start or top panel (depending on the `orientation` attribute value). For example, if the `orientation` attribute is set to `horizontal`, the `splitterPosition` attribute is 200 and the `positionedFromEnd` attribute is `false` (all default values), then ADF Faces places the splitter 200 pixels from the start panel, as shown in [Figure 9-23](#).

Figure 9-23 Splitter Position Measured from Start Panel



If the `positionedFromEnd` attribute is set to `true`, then ADF Faces measures the initial position of the splitter from the end (or bottom panel, depending on the `orientation` value). [Figure 9-24](#) shows the position of the splitter measured 200 pixels from the end panel.

Figure 9-24 Splitter Position Measured from End Panel



5. Set **collapsed** to determine whether or not the splitter is in a collapsed (hidden) state. By default, the `collapsed` attribute is `false`, which means both panes are

displayed. When the user clicks the arrow button on the splitter, the `collapsed` attribute is set to `true` and one of the panes is hidden.

ADF Faces uses the `collapsed` and `positionedFromEnd` attributes to determine which panel (that is, the first or second panel) to hide (collapse) when the user clicks the arrow button on the splitter. When the `collapsed` attribute is set to `true` and the `positionedFromEnd` attribute is `false`, the first panel is hidden and the second panel stretches to fill up the available space. When the `collapsed` attribute is `true` and the `positionedFromEnd` attribute is `true`, the second panel is hidden instead. Visually, the user can know which panel will be collapsed by looking at the direction of the arrow on the button: when the user clicks the arrow button on the splitter, the panel collapses in the direction of the arrow.

6. By default, the `panelSplitter` component stretches to fill available browser space. If you want to place the `panelSplitter` into a component that does not stretch its children, then you need to change how the `panelSplitter` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

In the Properties window, set **DimensionsFrom** to one of the following:

- `children`: Instead of stretching, the `panelSplitter` component will get its dimensions from its child component.

 **Note:**

If you use this setting and you set the `orientation` attribute to `vertical`, then the contents of the *collapsible* panel will not be determined by its child component, but instead will be determined by the value of `splitterPosition` attribute. The size of the other pane will be determined by its child component.

Additionally, you cannot set the height of the `panelSplitter` component (for example through the `inlineStyle` or `styleClass` attributes) if you use this setting. Doing so would cause conflict between the `panelSplitter` height and the child component height.

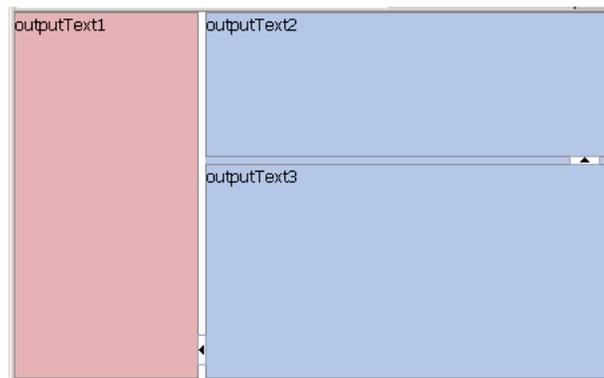
- `parent`: The size of the `panelSplitter` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
 - `auto`: If the parent component to the `panelSplitter` component allows stretching of its child, then the `panelSplitter` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelSplitter` component will be based on the size of its child component.
7. To place content in the component, drag and drop the desired component into the `first` and `second` facets. When you have the orientation set to `horizontal`, the `first` facet is the left facet. When you have the orientation set to `vertical`, the `first` facet is the top facet. If you want the child component to stretch, it must be a component that supports stretching. See [What You May Need to Know About Geometry Management and the `panelSplitter` Component](#).

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

 **Tip:**

If any facet is not visible in the visual editor:

- a. Right-click the `panelSplitter` component in the Structure window.
 - b. From the context menu, choose **Facets - Panel Splitter >facet *name***. Facets in use on the page are indicated by a checkmark in front of the facet name.
8. To create more than two panes, insert another **Panel Splitter** component into a facet to create nested splitter panes, as shown in [Figure 9-25](#).

Figure 9-25 Nested panelSplitter Components

The following example shows the code generated by JDeveloper when you nest splitter components.

```
<af:panelSplitter ...>
  <f:facet name="first">
    <!-- first panel child components components here -->
  </f:facet>
  <f:facet name="second">
    <!-- Contains nested splitter component -->
    <af:panelSplitter orientation="vertical" ...>
      <f:facet name="first">
        <!-- first panel child components components here -->
      </f:facet>
      <f:facet name="second">
        <!-- second panel child components components here -->
      </f:facet>
    </af:panelSplitter>
  </f:facet>
</af:panelSplitter>
```

9. If you want to perform some operation when users collapse or expand a panel, attach a client-side JavaScript using the `clientListener` tag for the `collapsed` attribute and a `propertyChange` event type. For information about client-side events, see [Handling Events](#).

What You May Need to Know About Geometry Management and the panelSplitter Component

The `panelSplitter` component can stretch its child components and it can also be stretched. The following components can be stretched inside the `first` or `second` facet of the `panelSplitter` component:

- `decorativeBox` (when configured to stretch)
- `deck`
- `calendar`
- `inputText` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox` (when configured to stretch)

- `panelCollection` (when configured to stretch)
- `panelDashboard` (when configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelHeader` (when configured to stretch)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `showDetailHeader` (when configured to stretch)
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a facet of the `panelSplitter` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch into facets of a component that stretches its child components. Therefore, if you need to place one of the components that cannot be stretched into a facet of the `panelSplitter` component, wrap that component in a transition component that does not stretch its child components.

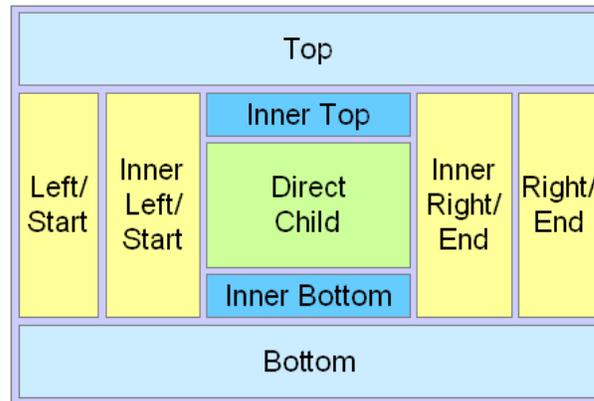
For example, if you want to place content in a `panelBox` component and have it flow within a facet of the `panelSplitter` component, you could place a `panelGroupLayout` component with its `layout` attribute set to `scroll` in a facet of the `panelSplitter` component, and then place the `panelBox` component in that `panelGroupLayout` component. See [Nesting Components Inside Components That Allow Stretching](#).

Arranging Page Contents in Predefined Fixed Areas

Using the ADF Faces `panelBorderLayout` component, you can arrange the content in fixed areas. This component does not stretch even if it is placed in a component that stretches its child components.

The `panelBorderLayout` component uses facets to contain components in predefined areas of a page. Instead of a `center` facet, the `panelBorder` layout component takes 0 to *n* direct child components (also known as indexed children), which are rendered consecutively in the center. The facets then surround the child components.

[Figure 9-26](#) shows the facets of the `panelBorderLayout` component: `top`, `inner top`, `bottom`, `inner bottom`, `start`, `inner start`, `end`, and `inner end`.

Figure 9-26 Facets in `panelBorderLayout`

The 12 supported facets of the `panelBorderLayout` component are:

- `top`: Renders child components above the center area.
- `bottom`: Renders child components below the center area.
- `start`: Supports multiple reading directions. This facet renders child components on the left of the center area between `top` and `bottom` facet child components, if the reading direction of the client browser is left-to-right. If the reading direction is right-to-left, it renders child components on the right of the center area. When your application must support both reading directions, this facet ensures that the content will be displayed on the proper side when the direction changes. If you do not need to support both directions, then you should use either the `left` or `right` facet.
- `end`: Supports multiple reading directions. This facet renders child components on the right of the center area between `top` and `bottom` facet child components, if the reading direction of the client browser is left-to-right. If the reading direction is right-to-left, it renders child components on the left of the center area. When your application must support both reading directions, this facet ensures that the content will be displayed on the proper side when the direction changes. If you do not need to support both directions, then you should use either the `left` or `right` facet.
- `left`: Supports only one reading direction. This facet renders child components on the left of the center area between `top` and `bottom` facet child components. When the reading direction is left-to-right, the `left` facet has precedence over the `start` facet if both the `left` and `start` facets are used (that is, contents in the `start` facet will not be displayed). If the reading direction is right-to-left, the `left` facet also has precedence over the `end` facet if both `left` and `end` facets are used.
- `right`: Supports only one reading direction. This facet renders child components on the right of the center area between `top` and `bottom` facet child components. If the reading direction is left-to-right, the `right` facet has precedence over the `end` facet if both `right` and `end` facets are used. If the reading direction is right-to-left, the `right` facet also has precedence over the `start` facet, if both `right` and `start` facets are used.
- `innerTop`: Renders child components above the center area but below the `top` facet child components.

- `innerBottom`: Renders child components below the center area but above the `bottom` facet child components.
- `innerLeft`: Renders child components similar to the `left` facet, but renders between the `innerTop` and `innerBottom` facets, and between the `left` facet and the center area.
- `innerRight`: Renders child components similar to the `right` facet, but renders between the `innerTop` facet and the `innerBottom` facet, and between the `right` facet and the center area.
- `innerStart`: Renders child components similar to the `innerLeft` facet, if the reading direction is left-to-right. Renders child components similar to the `innerRight` facet, if the reading direction is right-to-left.
- `innerEnd`: Renders child components similar to the `innerRight` facet, if the reading direction is left-to-right. Renders child components similar to the `innerLeft` facet, if the reading direction is right-to-left.

The `panelBorderLayout` component does not support stretching its child components, nor does it stretch when placed in a component that stretches its child components. Therefore, the size of each facet is determined by the size of the component it contains. If instead you want the contents to stretch to fill the browser window, consider using the `panelStretchLayout` component instead. See [Arranging Contents to Stretch Across a Page](#).

How to Use the `panelBorderLayout` Component to Arrange Page Contents in Predefined Fixed Areas

There is no restriction to the number of `panelBorderLayout` components you can have on a JSF page.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Arranging Page Contents in Predefined Fixed Areas](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelBorderLayout` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Border Layout** onto the JSF page.
2. From the Components window, drag and drop the component that will be used to display contents in the center of the window as a child component to the `panelBorderLayout` component.

Child components are displayed consecutively in the order in which you inserted them. If you want some other type of layout for the child components, wrap the components inside the `panelGroupLayout` component. See [Grouping Related Items](#).

3. To place contents that will surround the center, drag and drop the desired component into each of the facets.

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a

container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

 **Tip:**

If any facet is not visible in the visual editor:

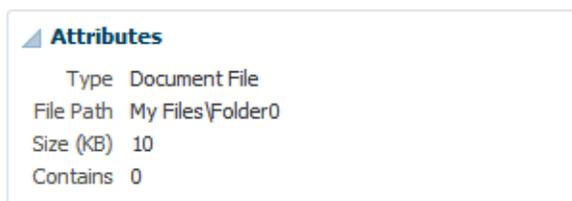
- a. Right-click the `panelBorderLayout` component in the Structure window.
- b. From the context menu, choose **Facets - Panel Border Layout >facet name**. Facets in use on the page are indicated by a checkmark in front of the facet name.

Arranging Content in Forms

Using the ADF Faces `panelFormLayout` component, you can display the fields and their labels in one or more columns. You can group the child components within the `panelFormLayout` component.

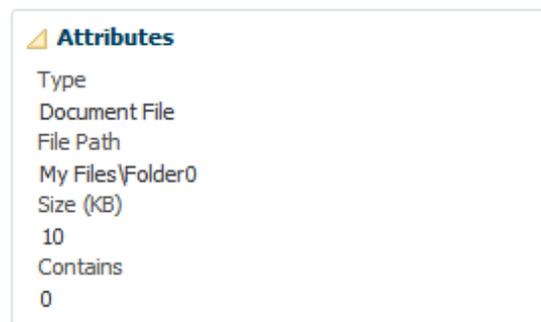
The `panelFormLayout` component lets you lay out multiple components such as input fields and selection list fields in one or more columns. The File Explorer application uses a `panelFormLayout` component to display file properties. The component is configured to have the labels right-aligned, as shown in [Figure 9-27](#).

Figure 9-27 Right-Aligned Labels and Left-Aligned Fields in a Form



[Figure 9-28](#) shows the same page with the component configured to display the labels above the fields.

Figure 9-28 Labels Above Fields in a Form



You can configure the `panelFormLayout` component to display the fields with their labels in one or more columns. Each field in the form is a child component of the `panelFormLayout` component. You set the desired number of rows, and if there are more child components than rows, the remaining child components are placed in a new column. The following example shows a `panelFormLayout` component with 10 `inputText` child components.

```
<af:panelFormLayout id="pf11" rows="10">
  <af:inputText label="Label 1" id="it1"/>
  <af:inputText label="Label 2" id="it2"/>
  <af:inputText label="Label 3" id="it3"/>
  <af:inputText label="Label 4" id="it4"/>
  <af:inputText label="Label 5" id="it5"/>
  <af:inputText label="Label 6" id="it6"/>
  <af:inputText label="Label 7" id="it7"/>
  <af:inputText label="Label 8" id="it8"/>
  <af:inputText label="Label 9" id="it9"/>
  <af:inputText label="Label 10" id="it10"/>
</af:panelFormLayout>
```

Because the `panelFormLayout`'s `row` attribute is set to 10, all 10 `inputText` components appear in one column, as shown in [Figure 9-29](#).

Figure 9-29 All `inputText` Components Display in One Column

Label 1

Label 2

Label 3

Label 4

Label 5

Label 6

Label 7

Label 8

Label 9

Label 10

However, if the `row` attribute were to be set to 8, then the first 8 `inputText` components display in the first column and the last two appear in the second column, as shown in [Figure 9-30](#).

Figure 9-30 Components Displayed in Two Columns

Label 1 Label 9

Label 2 Label 10

Label 3

Label 4

Label 5

Label 6

Label 7

Label 8

However, the number of rows displayed in each is not solely determined by the configured number of rows. By default, the `panelFormLayout` component's `maxColumns`

attribute is set to render no more than three columns (two for PDA applications). This value is what actually determines the number of rows. For example, if you have 25 child components and you set the component to display 5 rows and you leave the default maximum number of columns set to 3, then the component will actually display 9 rows, even though you have it set to display 5. This is because the maximum number of columns can override the set number of rows. Because it is set to allow only up to 3 columns, the component must use 9 rows in order to display all child components. You would need to set the maximum number of columns to 5 in order to have the component display just 5 rows.

ADF Faces uses default label and field widths, as determined by the standard HTML flow in the browser. You can also specify explicit widths to use for the labels and fields. Regardless of the number of columns in the form layout, the widths you specify apply to all labels and fields. You specify the widths using either absolute numbers in pixels or percentage values. If the length of a label does not fit, the text is wrapped.

 **Tip:**

If your page will be displayed in languages other than English, you should leave extra space in the labels to account for different languages and characters.

How to Use the `panelFormLayout` Component

You can use one or more `panelFormLayout` components on a page to create the desired form layout.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Arranging Content in Forms](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use `panelFormLayout`:

1. In the Components window, from the Layout panel, drag and drop a **Panel Form Layout** onto the JSF page.
2. In the Properties window, expand the Common section and set the label alignment.

By default, field labels on the child input components are displayed beside the fields. To place the labels above the fields, set the `labelAlignment` attribute to `top`.

 **Note:**

When you nest a `panelFormLayout` component inside another `panelFormLayout` component, the label alignment in the nested layout is `top`.

3. Set **rows** and **maxColumns** to determine the number of rows and columns in the `panelFormLayout` component.

The `rows` attribute value is the number that ADF Faces uses as the number of rows after which a new column will start. By default, it is set to 2147483647 (`Integer.MAX_VALUE`). This means all the child components that are set to `rendered="true"` and `visible="true"` will render in one, single column.

If you want the form to contain more than one column, set the `rows` attribute to a multiple of the number of rendered child components, and then set the `maxColumns` attribute to the maximum amount of columns that the form should display. The default value of `maxColumns` is 3. (On PDAs, the default is 2).

 **Note:**

If the `panelFormLayout` component is inside another `panelFormLayout` component, the inner `panelFormLayout` component's `maxColumns` value is always 1.

For example, if the `rows` attribute is set to 6 and there are 1 to 6 rendered child components, the list will be displayed in 1 column. If there are 7 to 12 rendered child components, the list will be displayed in 2 columns. If there are 13 or more child components, the list will be displayed in 3 columns. To display all rendered child components in 1 column, set the `rows` attribute back to the default value.

If the number of rendered child components would require more columns than allowed by the `maxColumn` attribute, then the value of the `rows` attribute is overridden. For example, if there are 100 rendered child components, and the `rows` attribute is set to 30 and the `maxColumns` attribute is 3 (default), the list will be displayed in 3 columns and 34 rows. If the `maxColumns` attribute is set to 2, the list will be displayed in 2 columns and 51 rows.

 **Tip:**

Rendered child components refers only to direct child components of the `panelFormLayout` component. Therefore, when a component that renders multiple rows (for example `selectManyCheckbox`) is a child, all its rows will be treated as a single rendered child and cannot be split across separate columns.

4. Set **fieldWidth**, **labelWidth**, and **layout** as needed.

ADF Faces uses default label and field widths, as determined by standard HTML flow in the browser. You can also specify explicit widths to use for the labels and fields.

The `labelWidth` attribute on the `panelFormLayout` component lets you set the preferred width for labels; the `fieldWidth` attribute lets you set the preferred width for fields. The `layout` attribute lets you set how the content is spaced out and if long labels and fields will be truncated.

 **Note:**

Any value you specify for the `labelWidth` component is ignored in layouts where the `labelAlignment` attribute is set to `top`, that is, in layouts where the labels are displayed above the fields.

Regardless of the number of columns in the form layout, the widths you specify apply to all labels and fields, that is, you cannot set different widths for different columns. You specify the widths using any CSS unit such as `em`, `px`, or `%`. The unit used must be the same for both the `labelWidth` and `fieldWidth` attribute.

When using percentage values:

- The percentage width you specify is a percent of the entire width taken up by the `panelFormLayout` component, regardless of the number of columns to be displayed.
- The sum of the `labelWidth` and `fieldWidth` percentages must add up to 100%. If the sum is less than 100%, the widths will be normalized to equal 100%. For example, if you set the `labelWidth` to 10% and the `fieldWidth` to 30%, at runtime the `labelWidth` would be 33% and the `fieldWidth` would be 67%.
- If you explicitly set the width of one but not the other (for example, you specify a percentage for `labelWidth` but not `fieldWidth`), ADF Faces automatically calculates the percentage width that is not specified.

 **Note:**

If your `panelFormLayout` component contains multiple columns and a footer, you may see a slight offset between the positioning of the main form items and the footer items in web browsers that do not honor fractional divisions of percentages. To minimize this effect, ensure that the percentage `labelWidth` is evenly divisible by the number of columns.

Suppose the width of the `panelFormLayout` component takes up 600 pixels of space, and the `labelWidth` attribute is set at 50%. In a one-column display, the label width will be 300 pixels and the field width will be 300 pixels. In a two-column display, each column is 300 pixels, so each label width in a column will be 150 pixels, and each field width in a column will be 150 pixels.

The `layout` attribute can be set to either `weighted`, which is the default value, or `fixed`. If the length of the label text does not fit on a single line with the given label width, ADF Faces automatically wraps the label text. If the given field width is less than the minimum size of the child content you have placed inside the `panelFormLayout` component, ADF Faces automatically uses the minimum size of the child content as the field width. However, if the `layout` attribute is set to `fixed`, labels and fields are not wrapped. Instead, they are truncated with CSS ellipses styling and an automatic title attribute that displays the full text on hover.

 **Note:**

If the field is wider than the space allocated, the browser will not truncate the field but instead will take space from the label columns. This potentially could cause the labels to wrap more than you would like. In this case, you may want to consider reducing the width of the field contents (for example, use a smaller `contentType` width on an `inputText` component). You could also set the `layout` attribute to `fixed`, which will truncate long labels to fit the allocated space instead of wrapping them.

5. Insert the desired child components.

Usually you insert labeled form input components, such as **Input Text**, **Select Many Checkbox**, and other similar components that enable users to provide input.

 **Tip:**

The `panelFormLayout` component also allows you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the `panelFormLayout` component.

The following example shows the `panelFormLayout` component as it is used on the `properties.jspx` page of the File Explorer application, shown in [Figure 9-27](#).

```
<af:panelFormLayout rows="5" id="pfl1">
  <af:inputText value="#{fileItemProperties.type}"
    label="#{explorerBundle['fileproperties.type']}"
    readOnly="true" id="it2"/>
  <af:inputText value="#{fileItemProperties.location}"
    label="#{explorerBundle['fileproperties.currentpath']}"
    readOnly="true" id="it3"/>
  <af:inputText value="#{fileItemProperties.size}"
    label="#{explorerBundle['fileproperties.size']}"
    readOnly="true" id="it4"/>
  <af:inputText value="#{fileItemProperties.contains}"
    label="#{explorerBundle['fileproperties.contains']}"
    readOnly="true" id="it5"/>
</af:panelFormLayout>
```

 **Tip:**

If you use components other than input components (which do not have `label` attributes) or if you want to group several input components with one single label inside a `panelFormLayout` component, first wrap the components inside a `panelLabelAndMessage` component.

6. To group semantically related input components in a form layout, use the `group` component to wrap those components that belong in a group. Components placed within a group will by default, cause the `panelFormLayout` component to draw a

separator lines at the beginning and end of the group. You can configure the `group` component so that the separator lines will always display, or will never display by setting the `StartBoundary` and `EndBoundary` attributes.

7. To add content below the child input components, insert the desired component into the `footer` facet.

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

The following example shows sample code that uses the `panelGroupLayout` component to arrange `footer` child components in a `panelFormLayout` component.

```
<af:panelFormLayout>
  <f:facet name="footer">
    <af:panelGroupLayout layout="horizontal">
      <af:button text="Save"/>
      <af:button text="Cancel"/>
      <f:facet name="separator">
        <af:spacer width="3" height="3"/>
      </f:facet>
    </af:panelGroupLayout>
  </f:facet>
  .
  .
  .
</af:panelFormLayout>
```

For information about using the `panelLabelAndMessage` component, see [Grouping Components with a Single Label and Message](#).

For information about using the `group` component, see [What You May Need to Know About Using the group Component with the panelFormLayout Component](#).

What You May Need to Know About Responsive Mode in the `panelFormLayout` Component

You can configure the `panelFormLayout` component to be responsive so that it modifies the form layout dynamically, depending on the space available. Responsive form layout allows dynamically adjusting the number of columns in which the children should be laid out, determining the label position of `panelFormLayoutComponent` children based on the space available for the component. To use `panelFormLayout` in responsive mode, set the `layout` attribute to `responsive`.

```
<af:panelFormLayout id="panelFormLayout1" layout="responsive">
  ...
</af:panelFormLayout>
```

The `panelFormLayout` component supports four panel dimensions: `panel-size-sm` (small), `panel-size-md` (medium), `panel-size-lg` (large), and `panel-size-xl` (extra large). By default, the panel uses one of these modes when the space available meets any of the following conditions:

- `sm` - <768px
- `md` - >=768px and < 1024px

- lg - $\geq 1024\text{px}$ and $< 1281\text{px}$
- xl - $\geq 1281\text{px}$

You can configure different set of values for the panel sizes by using the skinning property `tr-panel-size-[sm/md/lg]` in the ADF skin file. A panel dimension that is greater than the lg dimension limit is automatically an xl dimension.

```
af|panelFormLayout {
  -tr-panel-size-sm: 768;
  -tr-panel-size-md: 1024;
  -tr-panel-size-lg: 1281; /* anything greater than this is xl */
}
```

If the `panelFormLayout` component meets the panel dimensions described above, it will render a corresponding pseudo class on the component: `panel-size-sm`, `panel-size-md`, `panel-size-lg`, and `panel-size-xl`.

The number of columns used to lay out the `panelFormLayout` component children is controlled at runtime by the `maxColumns` attribute. The default value is 3. This attribute determines the maximum number of available columns to lay out the form fields, provided that sufficient space is available. If the available space reduces, the component will auto reduce the columns in which the form fields are laid out.



Note:

Extra wide fields like `af:inputText` with `row` attribute set to >1 should be put in the footer facet of the component in responsive mode. The fields in the footer are start aligned with fields in the first column of the form layout.

You can position the labels either as top-aligned or start-aligned. By default, the small panel mode will have form fields with top-aligned labels and all other panel modes will have start-aligned labels. You can customize the alignment behavior by using the skin aliases `AFTopAlignLabelCell:alias`, `AFTopAlignLabelContentCell:alias`, and `AFStartAlignLabelCell:alias` to align the start and top labels in different modes. The sample CSS to align labels is given below:

```
af|panelFormLayout:panel-size-sm::responsive-label-cell {
  -tr-rule-ref: selector(".AFTopAlignLabelCell:alias");
}
af|panelFormLayout:panel-size-sm .AFPpanelFormLayoutContentCell {
  -tr-rule-ref: selector(".AFTopAlignLabelContentCell:alias");
}

af|panelFormLayout:panel-size-md::responsive-label-cell,
af|panelFormLayout:panel-size-lg::responsive-label-cell,
af|panelFormLayout:panel-size-xl::responsive-label-cell {
  -tr-rule-ref: selector(".AFStartAlignLabelCell:alias");
}
```

By default, in start-aligned mode, label elements have 33.33% width and the content element have 66.66% width. You can change these values in the CSS. The sample CSS to set label and content width is given below:

```
af|panelFormLayout:panel-size-md::responsive-label-cell,
af|panelFormLayout:panel-size-lg::responsive-label-cell,
af|panelFormLayout:panel-size-xl::responsive-label-cell {
```

```

-tr-rule-ref: selector(".AFStartAlignLabelCell:alias");
width: 33.33%;
}
af|panelFormLayout:panel-size-md .AFPanelFormLayoutContentCell,
af|panelFormLayout:panel-size-lg .AFPanelFormLayoutContentCell,
af|panelFormLayout:panel-size-xl .AFPanelFormLayoutContentCell {
width: 66.66%;
}

```

What You May Need to Know About Using the group Component with the panelFormLayout Component

While the `group` component itself does not render anything, when you use it to group child components in the `panelFormLayout` component, by default, visible separators will display around the child components of each `group` component. For example, you might want to group some of the input fields in a form layout created by the `panelFormLayout` component. You can also choose to display a title for the group using its `title` attribute.

Note:

If the group title is not horizontally longer than the width of the current `panelFormLayout` column, the `panelFormLayout` will stretch horizontally to accommodate the title.

The `startBoundary` attribute controls whether or not the separator lines display at the top of the group, while the `endBoundary` attribute controls whether or not the separator lines display at the bottom of the group. If you want the line to display, set the attribute to `show`. If you don't want the lines to display, set the attribute to `hide`. In two adjacent groups, if you don't want the line to display, the adjoining attributes must both be set to `hide`, or one must be set to `hide` and the other to `dontCare`. By default, these attributes are set to `dontCare`, which means the parent component (in this case the `panelFormLayout` component) will display the lines.

The following sample code groups three sets of child components inside a `panelFormLayout` component. The first group is set to hide the separator lines. However, because the second group is configured to display a separator at the start of the group, the lines will display. The second group is also set to display a title and a line at the end of the group. Because the third group has the `startBoundary` attribute set to `dontCare`, the line at the bottom of the second group displays.

```

<af:panelFormLayout maxColumns="1" labelWidth="75" id="pfl4">
  <af:group id="g1" startBoundary="hide" endBoundary="hide">
    <af:selectOneChoice label="Prompt" value="option1" id="soc4">
      <af:selectItem label="Option 1" value="option1" id="si30"/>
      <af:selectItem label="Option 2" value="option2" id="si31"/>
    </af:selectOneChoice>
    <af:selectOneChoice label="Prompt" value="option1" id="soc5">
      <af:selectItem label="Option 1" value="option1" id="si32"/>
      <af:selectItem label="Option 2" value="option2" id="si33"/>
    </af:selectOneChoice>
    <af:panelLabelAndMessage label="Prompt" id="plam6" for="it6">
      <af:panelGroupLayout layout="horizontal" id="pgl4">

```

```

        <af:inputText simple="true" contentType="width: 100px;"
            label="inputText" id="it6"/>
        <af:button partialSubmit="true" text="Browse..." id="cb3"/>
    </af:panelGroupLayout>
</af:panelLabelAndMessage>
</af:group>
<af:group id="g2" title="Grouped Set of Forms" startBoundary="show"
    endBoundary="show">
    <af:selectManyListbox label="Prompt" contentType="width: 100px"
        id="sml3">
        <af:selectItem label="Option 1" value="option1" id="si34"/>
        <af:selectItem label="Option 2" value="option2" id="si35"/>
        <af:selectItem label="Option 3" value="option3" id="si36"/>
        <af:selectItem label="Option 4" value="option4" id="si37"/>
    </af:selectManyListbox>
</af:group>
<af:group id="g3" startBoundary="dontCare" endBoundary="dontCare">
    <af:selectManyCheckbox label="Prompt" id="smc3">
        <af:selectItem label="Value 1" value="value1" id="si38"/>
        <af:selectItem label="Value 2" value="value2" id="si39"/>
        <af:selectItem label="Value 3" value="value3" id="si40"/>
    </af:selectManyCheckbox>
</af:group>
</af:panelFormLayout>

```

At runtime the `panelFormLayout` component renders separator lines before and after the second `group` of child components, along with a title, as shown in [Figure 9-31](#).

Figure 9-31 Grouped Components in `panelFormLayout`

As described in [Arranging Content in Forms](#), the `panelFormLayout` component uses certain component attributes to determine how to display its child components (grouped and ungrouped) in columns and rows. When using the `group` component to group related components in a `panelFormLayout` component that will display its child components in more than one column, the child components of any `group` component

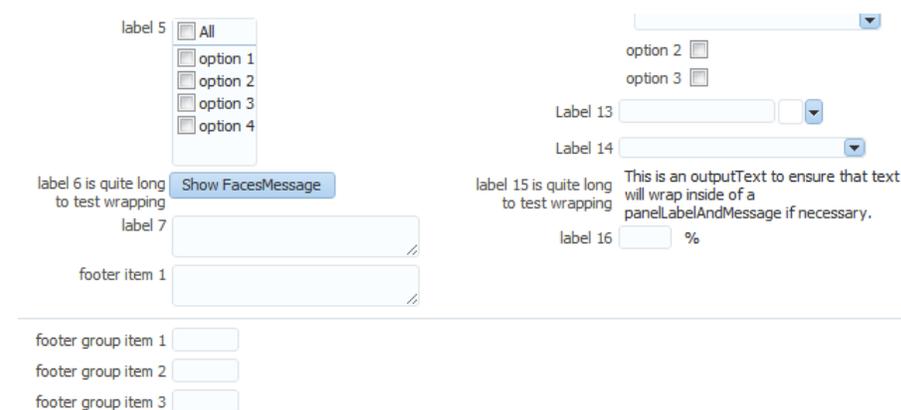
will always be displayed in the same column, that is, child components inside a `group` component will never be split across a column.

In JSP pages, facets can only contain one child component (Facelet pages do not have that restriction). Therefore, when you use the `group` component to group child components in the `footer` facet of the `panelFormLayout` component, you must place all the `group` components and other ungrouped child components in one root `group` component, as shown in the following example.

```
<af:panelFormLayout ...>
  <f:facet name="footer">
    <af:group id="g2">
      <af:inputText rows="2" label="footer item 1" id="it10"/>
      <af:group id="g3">
        <af:inputText columns="5" label="footer group item 1"
          id="it11"/>
        <af:inputText columns="5" label="footer group item 2"
          id="it12"/>
        <af:inputText columns="5" label="footer group item 3"
          id="it13"/>
      </af:group>
    </af:group>
    <af:panelGroupLayout layout="horizontal" id="pgl2">
      <f:facet name="separator">
        <af:spacer width="10" id="s2"/>
      </f:facet>
      <af:button text="Page 1" partialSubmit="true"
        id="cb3"/>
      <af:button text="Page 2" partialSubmit="true"
        id="cb4"/>
    </af:panelGroupLayout>
  </af:group>
</f:facet>
.
.
.
</af:panelFormLayout>
```

Like grouped child components in a `panelFormLayout` component, at runtime, by default the `panelFormLayout` component renders separator lines around the child components of each `group` component in the `footer` facet, as shown in [Figure 9-32](#).

Figure 9-32 Footer in panelGroupLayout with Grouped Components



 **Note:**

In JSP pages, the `footer` facet in the `panelFormLayout` component supports only two levels of grouped components, that is, you cannot have three or more levels of nested `group` components in the `footer` facet. For example, the following code is not valid:

```
<f:facet name="footer">
  <!-- Only one root group -->
  <af:group id="g1">
    <af:outputText value="Footer item 1" id="ot1"/>
    <!-- Any number of groups at this level -->
    <af:group id="g2">
      <af:outputText value="Group 1 item 1" id="ot2"/>
      <af:outputText value="Group 1 item 2" id="ot3"/>
      <!-- But not another nested group. This is illegal. -->
      <af:group id="g3">
        <af:outputText value="Nested Group 1 item 1" id="ot4"/>
        <af:outputText value="Nested Group 1 item 2" id="ot5"/>
      </af:group>
    </af:group>
  </af:group>
  <af:outputText value="Another footer item" id="ot6"/>
</f:facet>
```

When a `group` component is first in a column of a `panelFormLayout`, a separator line will not display at the top, even when `startBoundary` is set to `show`. The same is true for the last `group` component in a column; no separator line will display at the bottom, even if the `endBoundary` attribute is set to `show`.

Arranging Contents in a Dashboard

Use the ADF Faces `panelDashboard` component to use a `panelBox` component that contain content instead of fields. Based on the row height and column attribute, the `panelDashboard` arranges its child component and they stretch depending on the parent component.

The `panelDashboard` component allows you to arrange its child components in rows and columns, similar to the `panelForm` component. However, instead of text components, the `panelDashboard` children are `panelBox` components that contain content, as shown in [Figure 9-33](#).

Figure 9-33 panelDashboard with panelBox Child Components

When you add a `panelDashboard` component, you configure the number of columns it will contain, along with the height of each row. The dashboard stretches its children to fill up the configured space. If all the child components do not fit within the specified number of columns and row height, then the `panelDashboard` component displays a scroll bar.

When placed in a component that stretches its children, by default, the `panelDashboard` stretches to fill its parent container, no matter the number of children. This could mean that you may have blank space in the dashboard when the browser is resized to be much larger than the dashboard needs.

For example, say you have set the `panelDashboard` to inherit its size from its parent by setting the `dimensionsFrom` attribute to `parent`. You set `columns` to 1 and the `rowHeight` to 50px. You then add two `panelBox` components. Because `columns` is set to 1, you will have 2 rows. Because the parent component is a `panelStretchLayout`, the `panelDashboard` will stretch to fill the `panelStretchLayout`, no matter the height of the boxes, and you end up with extra space, as shown in [Figure 9-34](#) (the color of the dashboard has been changed to fuchsia to make it more easy to see its boundaries).

Figure 9-34 panelDashboard Stretches to Fill Space

If instead you don't want the dashboard to stretch, you can place it in a component that does not stretch its children, and you can configure the `panelDashboard` to determine its size based on its children (by setting the `dimensionsFrom` attribute to `children`). It will then be as tall as the number of rows required to display the children, multiplied by the `rowHeight` attribute.

In the previous example, if instead you place the dashboard in a `panelGroupLayout` set to `scroll`, because the `rowHeight` is set to 50, your `panelDashboard` will always be just over 100px tall, no matter the size of the browser window, as shown in [Figure 9-35](#).

Figure 9-35 `panelDashboard` Does Not Stretch



The `panelDashboard` component also supports declarative drag and drop behavior, so that the user can rearrange the child components. As shown in [Figure 9-36](#), the user can for example, move `panelBox 10` between `panelBox 4` and `panelBox 5`. A shadow is displayed where the box can be dropped.

Figure 9-36 Drag and Drop Capabilities in `panelDashboard`



 **Note:**

You can also configure drag and drop functionality that allows users to drag components into and out of the `panelDashboard` component. See [Adding Drag and Drop Functionality Into and Out of a `panelDashboard` Component](#).

Along with the ability to move child components, the `panelDashboard` component also provides an API that you can access to allow users to switch child components from being rendered to not rendered, giving the appearance of `panelBoxes` being inserted or deleted. The dashboard uses partial page rendering to redraw the new set of child components without needing to redraw the entire page.

You can use the `panelDashboardBehavior` tag to make the rendering of components appear more responsive. This tag allows the activation of a command component to apply visual changes to the dashboard before the application code modifies the component tree on the server. Because this opening up of space happens before the action event is sent to the server, the user will see immediate feedback while the action listener for the command component modifies the component tree and prepares the dashboard for the optimized encoding of the insert.

For example, [Figure 9-37](#) shows a `panelDashboard` component used in the right panel of a `panelSplitter` component. In the left panel, list items displayed as links represent each `panelBox` component in the `panelDashboard`. When all `panelBox` components are displayed, the links are all inactive. However, if a user deletes one of the `panelBox` components, the corresponding link becomes active. The user can click the link to reinsert the `panelBox`. By using the `panelDashboardBehavior` tag with the `commandLink` component, the user sees the inserted box drawing.

Figure 9-37 `commandLink` Components Use `panelDashboardBehavior` Tag

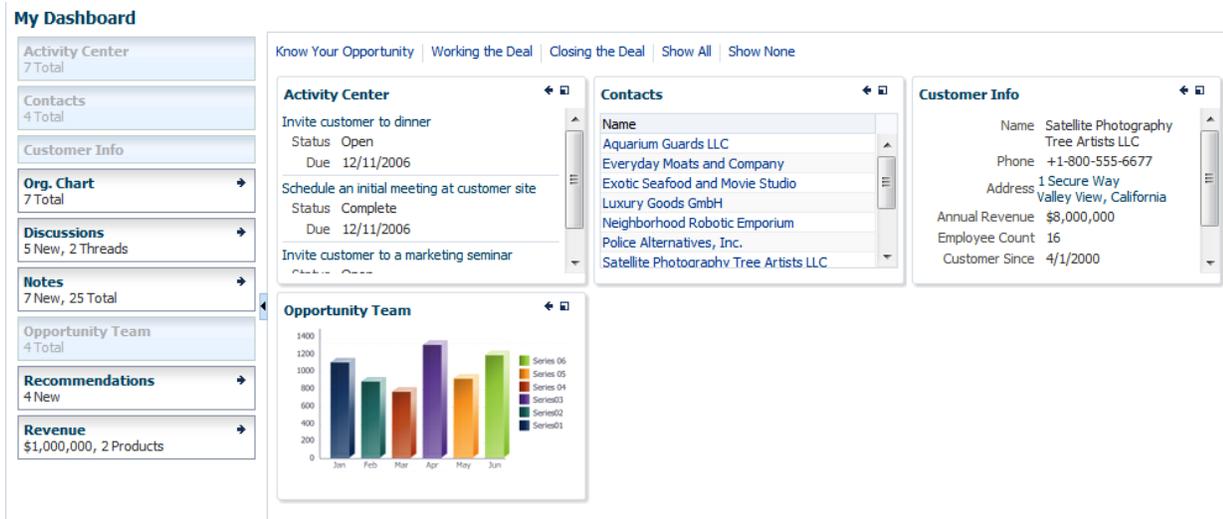


If you decide not to use this tag, there will be a slight delay while your action listener is processing before the user sees any change to the dashboard structure.

[Figure 9-38](#) shows a practical example using a `panelDashboard` component. Selecting one of the links at the top of the page changes the `panelBoxes` displayed in the

dashboard. The user can also add `panelBoxes` by clicking the associated link on the left-hand side of the page.

Figure 9-38 Practical Example of `panelDashboard`



How to Use the `panelDashboard` Component

After you add a `panelDashboard` to a page, you can configure the dashboard to determine whether or not it will stretch. Then, add child components, and if you want to allow rearrangement the components, also add a `componentDragSource` tag to the child component. If you want to allow insertion and deletion of components, implement a listener to handle the action. You can also use the `panelDashboardBehavior` tag to make the `panelDashboard` component appear more responsive to the insertion.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Arranging Contents in a Dashboard](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To use the `panelDashboard` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Dashboard** onto the page.
2. In the Properties window, expand the Common section.
3. Set **columns** to the number of columns you want to use to display the child components. The child components will stretch to fit each column.
4. Set **RowHeight** to the number of pixels high that each row should be. The child components will stretch to this height.
5. By default, the `panelDashboard` component stretches to fill available browser space. If instead, you want to use the `panelDashboard` component as a child to a component that does not stretch its children, then you need to change how the `panelDashboard` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

Expand the **Appearance** section, and set **DimensionsFrom** to one of the following:

- `children`: the `panelDashboard` component will get its dimensions from its child components.

 **Note:**

If you use this setting, you cannot set the height of the `panelDashboard` component (for example through the `inlineStyle` or `styleClass` attributes). Doing so would cause conflict between the `panelDashboard` height and the child component height.

- `parent`: the size of the `panelDashboard` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
 - `auto`: If the parent component to the `panelDashboard` component allows stretching of its child, then the `panelDashboard` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelDashboard` component will be based on the size of its child component.
6. From the Components window, drag and drop child `panelBox` components.

 **Tip:**

The `panelDashboard` component also supports the `region` component as a child component.

7. If you want users to be able to reorder the child components, in the Components window, from the Operations panel, in the Drag and Drop group, drag and drop a **Component Drag Source** as a child to each of the child components.
8. If you want to be able to add and delete components, create a managed bean and implement a handler method that will handle reordering children when a child is added or dropped. This event is considered a drop event, so you must use the Drag and Drop framework. For information about creating a handler for a drop event, see [Adding Drag and Drop Functionality](#).

To use the optimized lifecycle, have the handler call the `panelDashboard` component's `prepareOptimizedEncodingOfInsertedChild()` method, which causes the dashboard to send just the inserted child component to be rendered.

 **Note:**

If you plan on using the `panelDashboardBehavior` tag, then this API should be called from the associated command component's `actionListener` handler.

9. If you have added a `componentDragSource` tag in Step 7, then you must also implement a `DropEvent` handler for the `panelDashboard`. With the `panelDashboard` component selected, expand the **Behavior** section and bind the `DropListener` attribute to that handler method.
10. If you wish to use a `panelDashboardBehavior` tag, drag and drop a command component that will be used to initiate the insertion.
11. In the Properties window, bind the **ActionListener** for the command component to a handler on a managed bean that will handle the changes to the component tree. Have the handler call the `panelDashboard` component's `prepareOptimizedEncodingOfInsertedChild()` method, which causes the dashboard to send just the inserted child component to be rendered. The following example shows code on a managed bean that handles the insertion of child components.

```
public void handleInsert(ActionEvent e)
{
    UIComponent eventComponent = e.getComponent();
    String panelBoxId =
eventComponent.getAttributes().get("panelBoxId").toString();
    UIComponent panelBox = _dashboard.findComponent(panelBoxId);

    // Make this panelBox rendered:
    panelBox.setRendered(true);

    // Because the dashboard is already shown, perform an optimized
    // render so the whole dashboard does not have to be re-encoded:
    int insertIndex = 0;
    List<UIComponent> children = _dashboard.getChildren();
```

```

for (UIComponent child : children)
{
    if (child.equals(panelBox))
    {
        // Let the dashboard know that only the one child component should be
        // encoded during the render phase:
        _dashboard.prepareOptimizedEncodingOfInsertedChild(
            FacesContext.getCurrentInstance(),
            insertIndex);
        break;
    }

    if (child.isRendered())
    {
        // Count only rendered children because that is all that the
        // panelDashboard can see:
        insertIndex++;
    }
}
// Add the side bar as a partial target because we need to
// redraw the state of the side bar item that corresponds to the inserted item:
RequestContext rc = RequestContext.getCurrentInstance();
rc.addPartialTarget(_sideBar);
}

```

12. In the Components window, from the Operations panel, in the Behavior group, drag a **Panel Dashboard Behavior** tag and drop it as a child to the command component.
13. In the Properties window, enter the following:
 - **for:** Enter the ID for the associated `panelDashboard` component
 - **index:** Enter an EL expression that resolves to a method that determines the index of the component to be inserted. When you use the `panelDashboardBehavior` tag, a placeholder element is inserted into the DOM tree where the actual component will be rendered once it is returned from the server. Because the insertion placeholder gets added before the insertion occurs on the server, you must specify the location where you are planning to insert the child component so that if the user reloads the page, the children will continue to remain displayed in the same order.

What You May Need to Know About Geometry Management and the `panelDashboard` Component

This component organizes its children into a grid based on the number of columns and the `rowHeight` attribute. The child components that can be stretched inside of the `panelDashboard` include:

- `inputText` (when the `rows` attribute is set to greater than one, and the `simple` attribute is set to `true`)
- `panelBox`
- `region` (when configured to stretch)
- `table` (when configured to stretch)

If you try to put any other component as a child component to the `panelDashboard` component, then the component hierarchy is not valid.

Displaying and Hiding Contents Dynamically

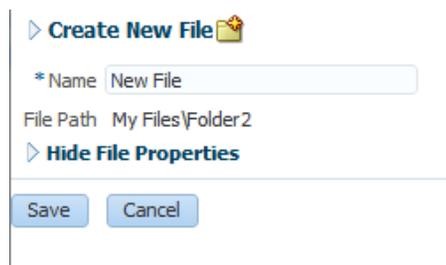
The ADF Faces `showDetail` component enables you to show or hide parts of the interface at will. You can also create a hierarchy of the content that you want to show or hide.

Sometimes you want users to have the choice of displaying or hiding content. When you do not need to show all the functionality of the user interface at once, you can save a lot of space by using components that enable users to show and hide parts of the interface at will.

The `showDetail` component creates a label with a toggle icon that allows users to disclose (show) or undisclose (hide) contents under the label. When the contents are undisclosed (hidden), the default label is **Show** and the expand icon is displayed. When the contents are disclosed (shown), the default label is **Hide**, and the collapse icon is displayed.

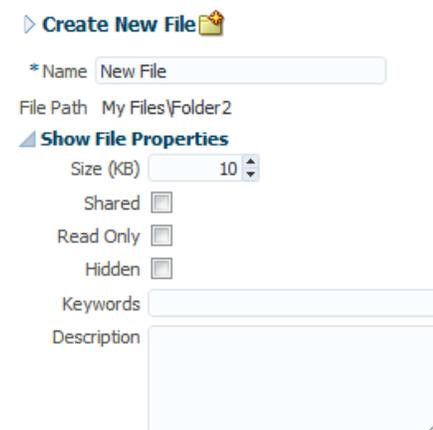
For example, the `newFileItem` page of the File Explorer application uses a `showDetail` component to hide and display file properties. The component is configured to hide the properties when the page is displayed, as shown in [Figure 9-39](#).

Figure 9-39 Collapsed showDetail



When the user clicks the toggle icon, the properties are displayed, as shown in [Figure 9-40](#).

Figure 9-40 Expanded showDetail



If you want to use something more complex than an `outputText` component to display the disclosed and undisclosed text, you can add components to the `showDetail` component's `prompt` facet. When set to be visible, any contents in the `prompt` facet will replace the disclosed and undisclosed text values. To use the `showDetail` component, see [How to Use the showDetail Component](#).

Like the `showDetail` component, the `showDetailHeader` component also toggles the display of contents, but the `showDetailHeader` component provides the label and toggle icon in a header, and also provides facets for a menu bar, toolbar, and text. Additionally, you can configure the `showDetailHeader` component to be used as a message for errors, warnings, information, or confirmations.

Tip:

The `showDetailHeader` component is the same as a `panelHeader` component, except that it handles disclosure events. For information about the `panelHeader` component, see [Displaying Items in a Static Box](#).

When there is not enough space to display everything in all the facets of the title line, the `showDetailHeader` text is truncated and displays an ellipsis, as shown in [Figure 9-41](#).

Figure 9-41 Text for the `showDetailHeader` Is Truncated



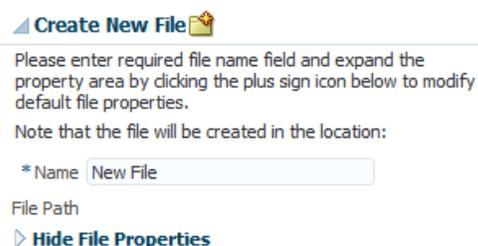
When there is more than enough room to display the contents, the extra space is placed between the `context` facet and the toolbar, as shown in [Figure 9-42](#).

Figure 9-42 Extra Space Is Added Before the Toolbar



The contents of the `showDetailHeader` component are undisclosed or disclosed below the header. For example, the `newFileItem` page of the File Explorer application uses a `showDetailHeader` component to display help for creating a new file. By default, the help is undisclosed, as shown in [Figure 9-40](#). When the user clicks the toggle icon in the header, the contents are disclosed, as shown in [Figure 9-43](#).

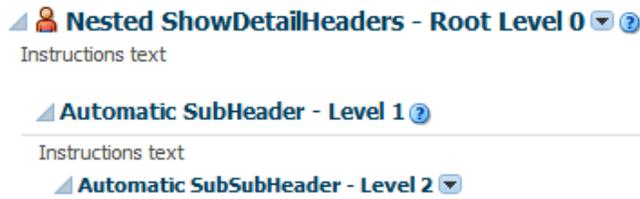
Figure 9-43 `showDetailHeader` Component Used to Display Help



You can also use the `showDetailHeader` component in conjunction with the `panelHeader` component to divide a page into sections and subsections, where some contents can be hidden. For information about the `panelHeader` component, see [Displaying Items in a Static Box](#).

You can nest `showDetailHeader` components to create a hierarchy of content. Each nested component takes on a different heading style to denote the hierarchy. [Figure 9-44](#) shows three nested `showDetailHeader` components, and their different styles.

Figure 9-44 Nested `showDetailHeader` Components Create a Hierarchy



 **Note:**

Heading sizes are determined by default by the physical containment of the header components. That is, the first header component will render as a heading level 1. Any header component nested in the first header component will render as a heading level 2, and so on. You can manually override the heading level on individual header components using the `headerLevel` attribute.

Use the `panelBox` component when you want information to be able to be displayed or hidden below the header, and you want the box to be offset from other information on the page. The File Explorer application uses two `panelBox` components on the `properties.jspx` page to display the attributes and history of a file, as shown in [Figure 9-45](#).

Figure 9-45 Two panelBox Components

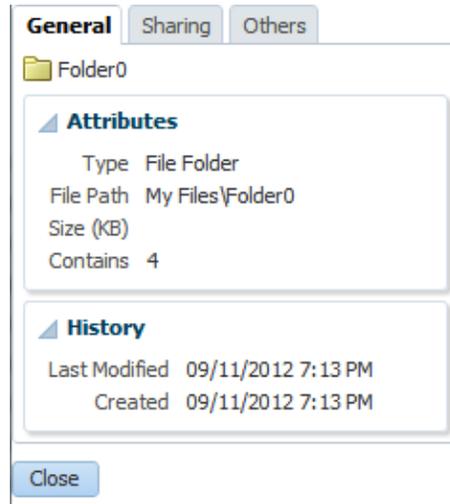
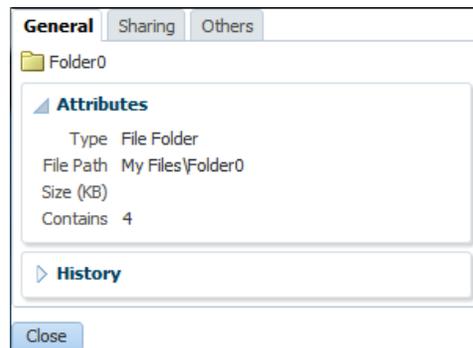
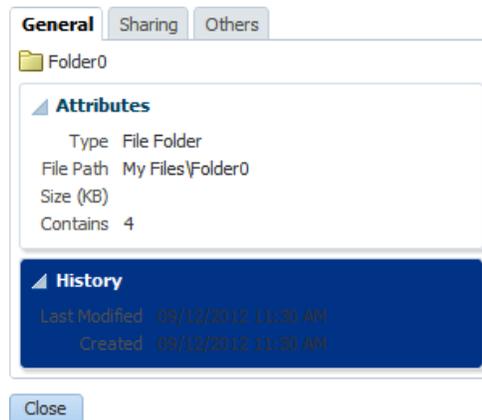


Figure 9-46 shows the same page, but with the History panelBox component in an undisclosed state.

Figure 9-46 Undisclosed panelBox Component



You can set the background color on a panelBox component so that the contents are further delineated from the rest of the page. Two color combinations (called ramps) are offered, and each combination contains four levels of color: none, light, medium, and dark. Figure 9-47 shows the same panel boxes as in Figure 9-45, but with the bottom panelBox component configured to show the medium tone of the core ramp.

Figure 9-47 Panel Boxes Using a Background Color

You can set the size of a `panelBox` component either explicitly by assigning a pixel size, or as a percentage of its parent. You can also set the alignment of the title, and add an icon. In addition, the `panelBox` component includes the `toolbar` facet that allows you to add a toolbar and toolbar buttons to the box.

You can control when the contents of an undisclosed component are sent and rendered to the client using the `contentDelivery` attribute. When set to `immediate` delivery, any undisclosed content is fetched during the initial request. With `lazy` delivery, the page initially goes through the standard lifecycle. However, instead of fetching the undisclosed content during that initial request, a special separate partial page rendering (PPR) request is run, and the undisclosed content is then returned. Because the page has just been rendered, only the Render Response phase executes for the undisclosed content, allowing the corresponding data to be fetched and displayed. You can configure it so that the contents are not rendered to the client until the first request to disclose the content and the contents then remain in the cache (`lazy`), or so that the contents are rendered each time there is a request to disclose them (`lazyUncached`).

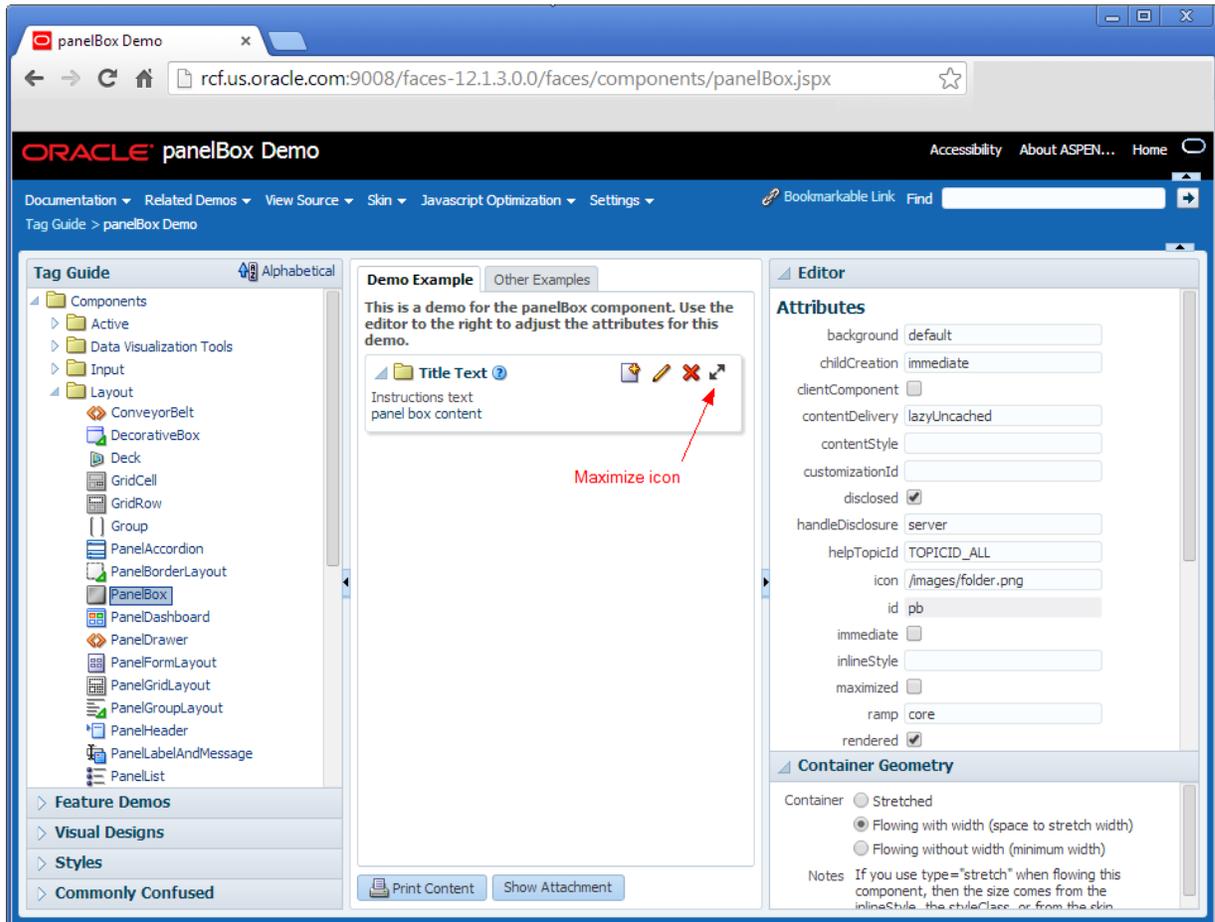
For all three of these components, you can use the `childCreation` attribute. This attribute affects JSP tag in determining when the `UIComponent` children are actually created. By default, all child components are created when the parent component is created. If you configure the component to use `lazy` or `lazyUncached`, the child components are not created when the parent tag is certain that a rendered instance of the component will be created. If there will be a large number of children, to improve performance you can configure these components so that they create the child components only when they are disclosed, or so that they create the child components only when they are disclosed the first time, and from that point on they remain created.

 **Note:**

The `childCreation` attribute only attempts to delay creation of the child components. When EL is used as the value for the `disclosed` attribute or the disclosure component is being stamped (inside of an iterator for example), the children will always be created, regardless of the `childCreation` attribute value.

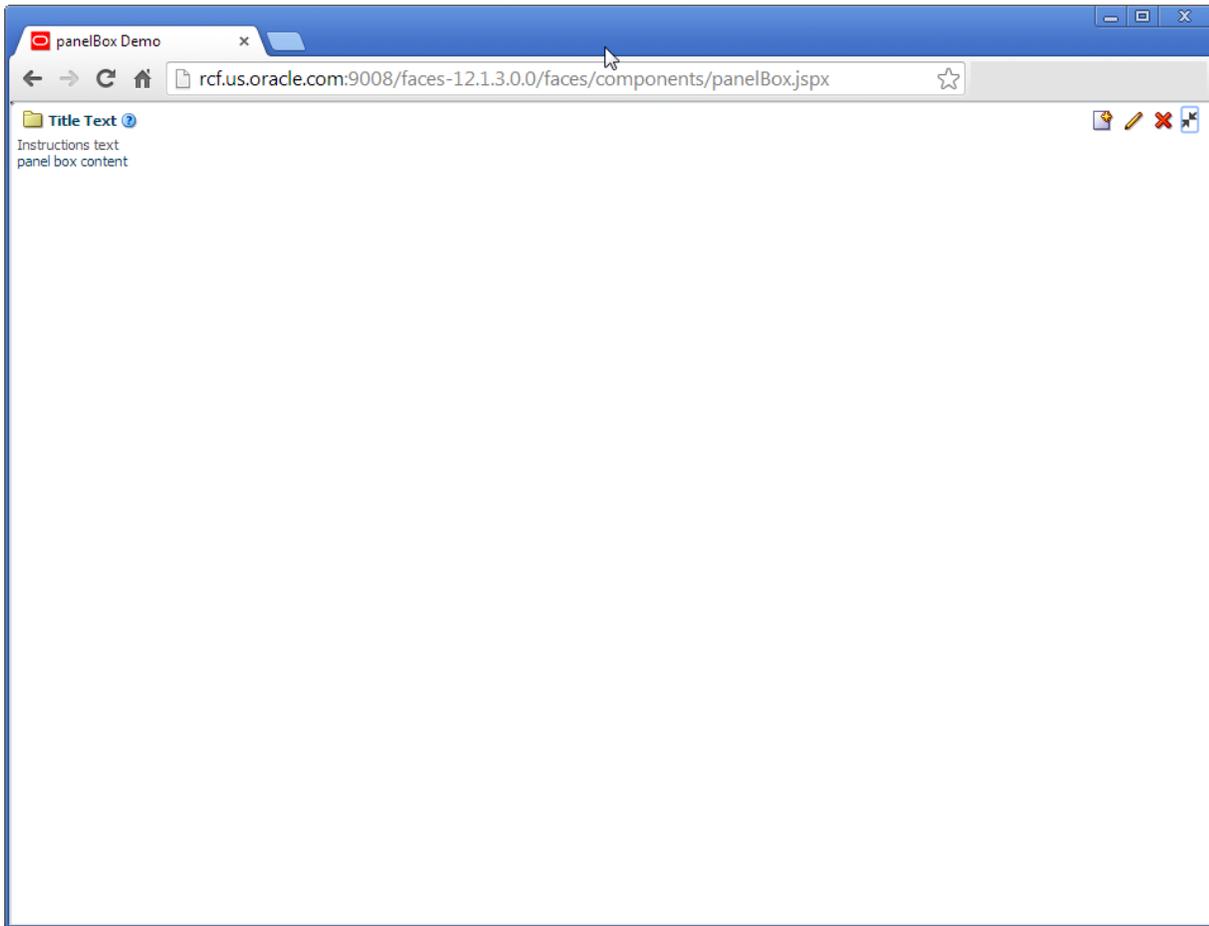
The `showDetailHeader` and the `panelBox` components both can be maximized to display in the full browser window. You can also configure an icon to display that allows the user to maximize and then restore the component to normal size. [Figure 9-48](#) shows the demo application with the `panelBox` component at its normal size. Notice the maximize icon in the header.

Figure 9-48 `panelBox` Demo with `panelBox` at Normal Size



When a user clicks the maximize icon, the `panelBox` is redrawn to take up the entire browser window, as shown in [Figure 9-49](#). The user can click the restore icon to return the component to its normal size.

Figure 9-49 Maximized panelBox component



By default, the component is configured to only show the maximize icon on tablet devices. On desktops, no icon is visible. You can also configure the component so that the icon is always displayed or never displayed. Additionally, you can create a listener that can be used to determine when to maximize the component.

If you want to show and hide multiple large areas of content, consider using the `panelAccordion` and `panelTabbed` components. See [Displaying or Hiding Contents in Panels](#).

How to Use the `showDetail` Component

Use the `showDetail` component to show and hide a single set of content.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying and Hiding Contents Dynamically](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `showDetail` component:

1. In the Components window, from the Layout panel, drag and drop a **Show Detail** onto the JSF page.
2. In the Properties window, expand the Common section and set the attributes as needed.

Set **Disclosed** to `true` if you want the component to show its child components.

 **Note:**

While the user can change the value of the `disclosed` attribute by displaying and hiding the contents, the value will not be retained once the user leaves the page unless you configure your application to allow user customizations. For information, see [Allowing User Customization on JSF Pages](#).

Set **DisclosedText** to the label you want to display next to the toggle icon when the contents are disclosed (shown). By default, the label is **Hide** if no value is specified.

Set **UndisclosedText** to the label you want to display next to the toggle icon when the contents are undisclosed (hidden). By default, the label is **Show** if no value is specified.

 **Note:**

If you specify a value for `disclosedText` but not for `undisclosedText`, then ADF Faces automatically uses the `disclosedText` value for both the disclosed state and undisclosed state. Similarly, if you specify a value for `undisclosedText` but not for `disclosedText`, the `undisclosedText` value is used when the contents are hidden or displayed.

Instead of using text specified in `disclosedText` and `undisclosedText`, you could use the `prompt` facet to add a component that will render next to the toggle icon.

You can also change the padding between the `showDetail` component and any child component. See [What You May Need to Know About Skinning and the showDetail Component](#).

3. Expand the Behavior section and set **DisclosureListener** to a `DisclosureListener` method in a backing bean that you want to execute when the user displays or hides the component's contents.

For information about disclosure events and listeners, see [What You May Need to Know About Disclosure Events](#).

4. You can configure when the child components will be created using the `childCreation` attribute. To do so, expand the **Other** section, and set **ChildCreation** to one of the following:
 - `immediate`: All child components are created when the `showDetail` component is created.

- `lazy`: The child components are created only when they are disclosed. Once disclosed and the associated child components are rendered, they remain created in the component tree.
 - `lazyUncached`: The child components are created only when they are disclosed. Once the components are hidden, they are destroyed.
5. You can configure when content of undisclosed children will be sent to the client using the `contentDelivery` attribute. To do so, expand the **Other** section, and set **ContentDelivery** to one of the following:
 - `immediate`: All undisclosed content is sent when the `showDetail` component is created.
 - `lazy`: The undisclosed content is sent only when the content is first disclosed. Once disclosed and the content is rendered, it remains in memory.
 - `lazyUncached`: The undisclosed content is created every time it is disclosed. If the content is subsequently hidden, it is destroyed.
 6. To add content, insert the desired child components inside the `showDetail` component.

How to Use the `showDetailHeader` Component

Use the `showDetailHeader` component when you want to display a single set of content under a header, or when you want the content to be used as messages that can be displayed or hidden. You can also use the `showDetailHeader` component to create a hierarchy of headings and content when you want the content to be able to be hidden.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying and Hiding Contents Dynamically](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `showDetailHeader` component:

1. In the Components window, from the Layout panel, drag and drop a **Show Detail Header** onto the JSF page.
2. In the Properties window, expand the Common section. Set **Text** to the text string you want for the section header label.
3. Set **Icon** to the URI of the image file you want to use for the section header icon. The icon image is displayed before the header label.

Note:

Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

4. If you are using the header to provide specific messaging information, set **MessageType** to one of the following values:
 - `confirmation`: The confirmation icon (represented by a note page overlaid with a green checkmark) replaces any specified icon image.
 - `error`: The error icon (represented by a red circle with an x inside) replaces any specified icon image. The header label also changes to red.
 - `info`: The info icon (represented by a blue circle with an *i* inside) replaces any specified icon image.
 - `warning`: The warning icon (represented by a yellow triangle with an exclamation mark inside) replaces any specified icon image.
 - `none`: Default. No icon is displayed, unless one is specified for the `icon` attribute.

Figure 9-50 shows each of the icons used for message types: a red X for an error, a yellow triangle with an exclamation point for a warning, a green check for a confirmation, and a blue circle with an "i" for information.

Figure 9-50 Icons Used for Message Types



 **Note:**

Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

5. Set **Disclosed** to `true` if you want the component to show its child components.

 **Note:**

If you set the value of the `disclosed` attribute to `false`, then any validation present for child components of `showDetailHeader` will not be performed, because the child components are not rendered.

While the user can change the value of the `disclosed` attribute by displaying and hiding the contents, the value will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).

6. Expand the Behavior section and set **DisclosureListener** to a `disclosureListener` method in a backing bean that you want to execute when the user displays or hides the component's contents.

For information about disclosure events and listeners, see [What You May Need to Know About Disclosure Events](#).

7. If you want to control how the `showDetailHeader` component handles geometry management, expand the Appearance section and set **Type**. Set it to `flow` if you do not want the component to stretch or to stretch its children. The height of the `showDetailHeader` component will be determined solely by its children. Set it to `stretch` if you want it to stretch and stretch its child (will only stretch a single child component). Leave it set to the default if you want the parent component of the `showDetailHeader` component to determine geometry management. For information about geometry management, see [Geometry Management and Component Stretching](#).
8. To add buttons or icons to the header, in the Components window, from the Layout panel, in the Menus and Toolbar Containers group, drag and drop the `toolbar` component into the `toolbar` facet. Then add any number of button components into the newly inserted `toolbar` component. For information about using the `toolbar` component, see [Using Toolbars](#).

 **Note:**

Toolbar overflow is not supported in `panelHeader` components.

9. To add menus to the header, insert menu components into the `menuBar` facet. For information about creating menus, see [Using Menus in a Menu Bar](#).

 **Tip:**

You can place menus in the `toolbar` facet and toolbars (and toolboxes) in the `menu` facet. The main difference between these facets is location. The `toolbar` facet is before the `menu` facet.

10. To create a subsection header, insert another `showDetailHeader` component inside an existing `showDetailHeader` component.

11. To override the heading level for the component, set **headerLevel** to the desired level, for example H1, H2, etc. through H6.

The heading level is used to determine the correct page structure, especially when used with screen reader applications. By default, `headerLevel` is set to -1, which allows the headers to determine their size based on the physical location on the page. In other words, the first header component will be set to be a H1. Any header component nested in that H1 component will be set to H2, and so on.

 **Note:**

Screen reader applications rely on the HTML header level assignments to identify the underlying structure of the page. Make sure your use of header components and assignment of header levels make sense for your page.

When using an override value, consider the effects of having headers inside disclosable sections of the page. For example, if a page has collapsible areas, you need to be sure that the overridden structure will make sense when the areas are both collapsed and disclosed.

12. If you want to change just the size of the header text, and not the structure of the heading hierarchy, set the `size` attribute.

The `size` attribute specifies the number to use for the header text and overrides the skin. The largest number is 0, and it corresponds to an H1 header level; the smallest is 5, and it corresponds to an H6 header.

By default, the `size` attribute is -1. This means ADF Faces automatically calculates the header level style to use from the topmost, parent component. When you use nested components, you do not have to set the `size` attribute explicitly to get the proper header style to be displayed.

 **Note:**

While you can force the style of the text using the `size` attribute, (where 0 is the largest text), the value of the `size` attribute will not affect the hierarchy. It only affects the style of the text.

You can also use skins to change the appearance of the different headers. For information, see [What You May Need to Know About Skinning and the `showDetailHeader` Component](#).

13. You can configure when the child components will be created using the `childCreation` attribute. To do so, expand the **Other** section, and set **ChildCreation** to one of the following:
 - `immediate`: All child components are created when the `showDetailHeader` component is created.
 - `lazy`: The child components are created only when they are disclosed. Once disclosed and the associated child components are rendered, they remain created in the component tree.

- `lazyUncached`: The child components are created only when they are disclosed. Once the components are hidden, they are destroyed.
14. You can configure when content of undisclosed children will be sent to the client using the `contentDelivery` attribute. To do so, expand the **Other** section, and set **ContentDelivery** to one of the following:
 - `immediate`: All undisclosed content is sent when the `showDetailHeader` component is created.
 - `lazy`: The undisclosed content is sent only when the content is first disclosed. Once disclosed and the content is rendered, it remains in memory.
 - `lazyUncached`: The undisclosed content is created every time it is disclosed. If the content is subsequently hidden, it is destroyed.
 15. If you want users to be able to maximize the `showDetailHeader` component so that it renders in the full browser window, in the **Other** section, set **ShowMaximized** to one of the following:
 - `always`: The maximize icon is always displayed.
 - `never`: The maximize icon is never displayed
 - `auto`: The maximize icon is displayed only on mobile devices. This is the default.

You can also programmatically set the `showDetailHeader` component to be maximized. You can use an EL expression as the value of the `maximized` attribute to resolve to `true`, or you can create a listener method that sets that attribute and listen for it using the `maximizeListener` attribute.

16. To add content to a section or subsection, insert the desired child components inside the `showDetailHeader` component.

How to Use the `panelBox` Component

You can insert any number of `panelBox` components on a page.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying and Hiding Contents Dynamically](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use a `panelBox` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Box** onto the JSF page.
2. In the Properties window, expand the Appearance section, and for **Ramp**, select the ramp you wish to use.

The `core` ramp uses variations of blue, while the `highlight` ramp uses variations of yellow. You can change the colors used by creating a custom skin. See [What You May Need to Know About Skinning and the `panelBox` Component](#).

3. Set **Background** to one of the following values: `light`, `medium`, `dark`, or `default`. The default background color is transparent.

4. Set **Text** to the text string you want to display as the title in the header portion of the container.
5. Set **Icon** to the URI of the icon image you want to display before the header text.

 **Note:**

If both the `text` and `icon` attributes are not set, ADF Faces does not display the header portion of the `panelBox` component.

 **Note:**

Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

6. Set **TitleAlign** to one of the following values: `center`, `start`, `end`, `left`, or `right`. The value determines the horizontal alignment of the title (including any icon image) in the header portion of the container.
7. Expand the Behavior section and set **DisclosureListener** to a `disclosureListener` method in a backing bean that you want to execute when the user shows or hides the component's contents.

For information about disclosure events and listeners, see [What You May Need to Know About Disclosure Events](#).

8. To add toolbar buttons, in the Components window, from the Layout panel, in the Menus and Toolbar Containers group, drag and drop a **Toolbar** into the `toolbar` facet. Then insert the desired number of button components into the `toolbar` component. For information about using `toolbar` and `button` components, see [Using Toolbars](#).

 **Tip:**

If any facet is not visible in the visual editor:

- a. Right-click the `panelBox` component in the Structure window.
- b. From the context menu, choose **Facets - Panel Box >Toolbar**. Facets in use on the page are indicated by a checkmark in front of the facet name.

9. To change the width of the `panelBox` component, set the `inlineStyle` attribute to the exact pixel size you want. Alternatively, you can set the `inlineStyle` attribute to a percentage of the outer element that contains the `panelBox` component. The following example shows the code you might use for changing the width.

```
<af:panelBox inlineStyle="width:50%;" ...>
  <!-- child contents here -->
</af:panelBox>
```

10. You can configure when the child components will be created using the `childCreation` attribute. To do so, expand the **Other** section, and set **ChildCreation** to one of the following:
 - `immediate`: All child components are created when the `panelBox` component is created.
 - `lazy`: The child components are created only when they are disclosed. Once disclosed and the associated child components are rendered, they remain created in the component tree.
 - `lazyUncached`: The child components are created only when they are disclosed. Once the components are hidden, they are destroyed.
11. You can configure when content of undisclosed children will be sent to the client using the `contentDelivery` attribute. To do so, expand the **Other** section, and set **ContentDelivery** to one of the following:
 - `immediate`: All undisclosed content is sent when the `panelBox` component is created.
 - `lazy`: The undisclosed content is sent only when the content is first disclosed. Once disclosed and the content is rendered, it remains in memory.
 - `lazyUncached`: The undisclosed content is created every time it is disclosed. If the content is subsequently hidden, it is destroyed.
12. If you want users to be able to maximize the `panelBox` component so that it renders in the full browser window, in the **Other** section, set **ShowMaximized** to one of the following:
 - `always`: The maximize icon is always displayed.
 - `never`: The maximize icon is never displayed
 - `auto`: The maximize icon is displayed only on mobile devices. This is the default.

You can also programmatically set the `panelBox` component to be maximized. You can use an EL expression as the value of the `maximized` attribute to resolve to `true`, or you can create a listener method that sets that attribute and listen for it using the `maximizeListener` attribute.

13. To add contents to the container for display, insert the desired components as child components to the `panelBox` component.

Typically, you would insert one child component into the `panelBox` component, and then insert the contents for display into the child component. The child component controls how the contents will be displayed, not the parent `panelBox` component.

What You May Need to Know About Disclosure Events

The `disclosed` attribute specifies whether to show (disclose) or hide (undisclose) the contents under its header. By default, the `disclosed` attribute is `true`, that is, the contents are shown. When the attribute is set to `false`, the contents are hidden. You do not have to write any code to enable the toggling of contents from disclosed to undisclosed, and vice versa. ADF Faces handles the toggling automatically.

When the user clicks the toggle icon to show or hide contents, by default, the components deliver a `org.apache.myfaces.trinidad.event.DisclosureEvent` event to the server. The `DisclosureEvent` event contains information about the source

component and its state: whether it is disclosed (expanded) or undisclosed (collapsed). The `isExpanded()` method returns a `boolean` value that determines whether to expand (disclose) or collapse (undisclose) the node. If you only want the component to disclose and undisclose its contents, then you do not need to write any code.

However, if you want to perform special handling of a `DisclosureEvent` event, you can bind the component's `disclosureListener` attribute to a `disclosureListener` method in a backing bean. The `disclosureListener` method will then be invoked in response to a `DisclosureEvent` event, that is, whenever the user clicks the disclosed or undisclosed icon.

The `disclosureListener` method must be a public method with a single `disclosureEvent` event object and a void return type, like the following:

```
public void some_disclosureListener(DisclosureEvent disclosureEvent) {  
    // Add event handling code here  
}
```

By default, `DisclosureEvent` events are usually delivered in the Invoke Application phase, unless the component's `immediate` attribute is set to `true`. When the `immediate` attribute is set to `true`, the event is delivered in the earliest possible phase, usually the Apply Request Values phase.

If you want to have a `disclosureListener` method and you also want to react to the event on the client, you can use the `AdfDisclosureEvent` client-side event. The event root for the client `AdfDisclosureEvent` event is set to the event source component: only the event for the panel whose `disclosed` attribute is `true` gets sent to the server. For information about client-side events and event roots, see [Handling Events](#).

The value of the `disclosed` attribute can be persisted at runtime, that is, when the user shows or hides contents, ADF Faces can change and then persist the attribute value so that it remains in that state for the length of the user's session. See [Allowing User Customization on JSF Pages](#).

 **Note:**

Any ADF Faces component that has built-in event functionality, as the `showDetail`, `showDetailHeader`, and `panelBox` components do, must be enclosed in the `form` component.

What You May Need to Know About Skinning and the `showDetail` Component

In the default skin used by ADF Faces, child components of the `showDetail` component are indented. You can control the indentation using the `child-container` skinning key. For example:

```
af|showDetail { -tr-layout: flush;}  
af|showDetail::child-container {  
    padding-left: 10px;  
}
```

See [Customizing the Appearance Using Styles and Skins](#).

What You May Need to Know About Skinning and the `showDetailHeader` Component

Also by default, the style used for heading sizes for the `showDetailHeader` component are controlled by the skin. Heading sizes above 2 will be displayed the same as size 2. That is, there is no difference in styles for sizes 3, 4, or 5—they all show the same style as size 2. You can change this by creating a custom skin.

See [Customizing the Appearance Using Styles and Skins](#).

What You May Need to Know About Skinning and the `panelBox` Component

The `core` ramp of the `panelBox` component uses variations of blue, while the `highlight` ramp uses variations of yellow. You can change the colors by creating a custom skin and configuring various `panelBox` skinning style selectors.

These style selectors are all augmented by the two pseudo-classes. The first pseudo-class is `ramp` which can have values of `:core` or `:highlight`. The second pseudo-class is `background` which can have the values of `:default`, `:light`, `:medium`, or `:dark`. For example, if you want the background color to be lime green on the content area when the `panelBox` `ramp` attribute is set to `core` and `background` is set to `default`, you could do the following:

```
af|panelBox::content:core:default {background-color: lime; border: none;}
```

You can also use the aliases to change the header and content. For example, `.AFPanelBoxContentCoreMedium:alias` is included in the `af|panelBox::content:core:medium` selector. So if you want to change the background color of the `core` `medium` `panelBox` content area, you can use the `.AFPanelBoxContentCoreMedium:alias` instead of using multiple pseudo-classes.

See [Customizing the Appearance Using Styles and Skins](#).

Displaying or Hiding Contents in Panels

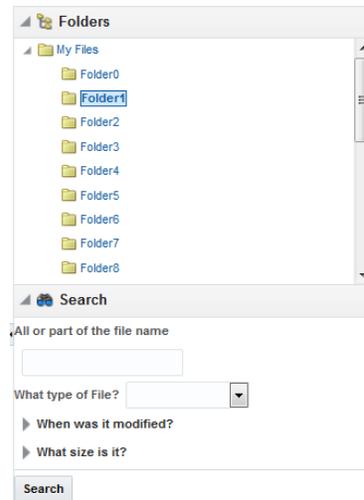
Using the ADF Faces `panelAccordion`, `panelTabbed`, `panelDrawer`, or `panelSpringboard` components, you can show or hide multiple areas of content. When you need to display multiple areas of content that can be hidden and displayed, you can use the `panelAccordion`, the `panelTabbed`, `panelDrawer`, or `panelSpringboard` components. These components use the `showDetailItem` component to display the actual contents.

The `panelAccordion` component creates a series of expandable panes. You can allow users to expand more than one panel at any time, or to expand only one panel at a time. When more than one panel is expanded, the user can adjust the height of the panel by dragging the header of the `showDetailItem` component.

When a panel is collapsed, only the panel header is displayed; when a panel is expanded, the panel contents are displayed beneath the panel header (users can expand the panes by clicking either the `panelAccordion` component's header or the

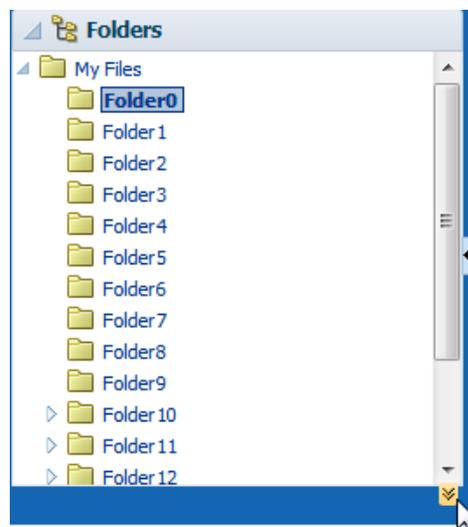
expand icon). The File Explorer application uses the `panelAccordion` component to display the Folders and Search panes, as shown in [Figure 9-51](#).

Figure 9-51 `panelAccordion` Panes



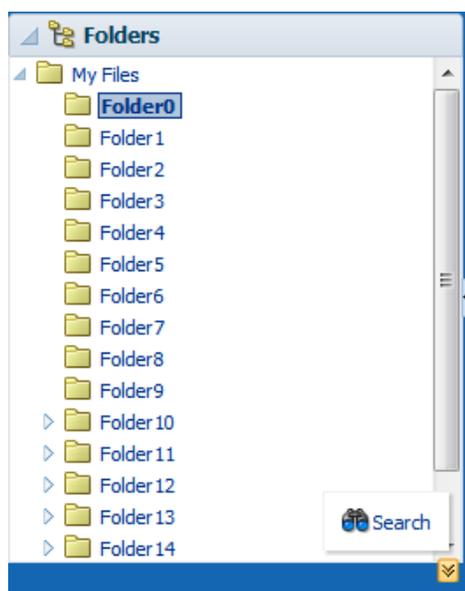
At runtime, when available browser space is less than the space needed to display expanded panel contents, ADF Faces automatically displays overflow icons that enable users to select and navigate to those panes that are out of view. [Figure 9-52](#) shows the overflow icon (a chevron) displayed in the lower right-hand corner of the Folders panel of the File Explorer application, when there is not enough room to display the Search panel.

Figure 9-52 Overflow Icon In `panelAccordion`

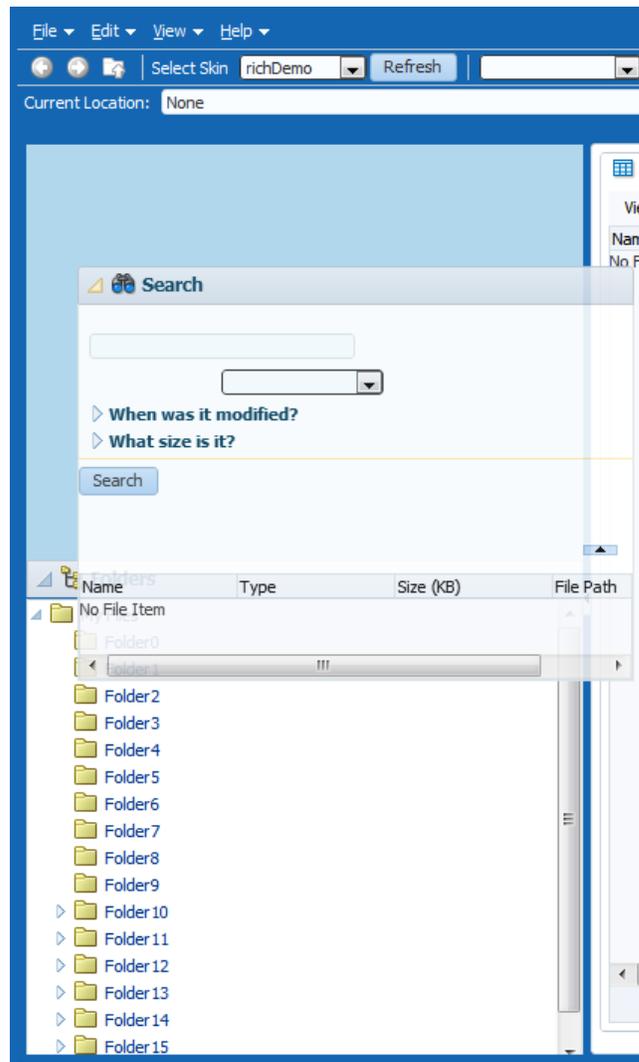


When the user clicks the overflow icon, ADF Faces displays the overflow popup menu (as shown in [Figure 9-53](#)) for the user to select and navigate to.

Figure 9-53 Overflow Popup Menu in panelAccordion



You can also configure the `panelAccordion` so that the panes can be rearranged by dragging and dropping, as shown in [Figure 9-54](#).

Figure 9-54 Panes Can Be Reordered by Dragging and Dropping

When the order is changed, the `displayIndex` attribute on the `showDetailItem` components also changes to reflect the new order.

 **Note:**

Items in the overflow cannot be reordered.

To use the `panelAccordion` component, see [How to Use the panelAccordion Component](#).

The `panelTabbed` component creates a series of tabbed panes. Unlike the `panelAccordion` panes, the `panelTabbed` panes are not collapsible or expandable. Instead, when users select a tab, the contents of the selected tab are displayed. The tabs may be positioned above, below, above and below (both), to the left, or to the right of the display area.

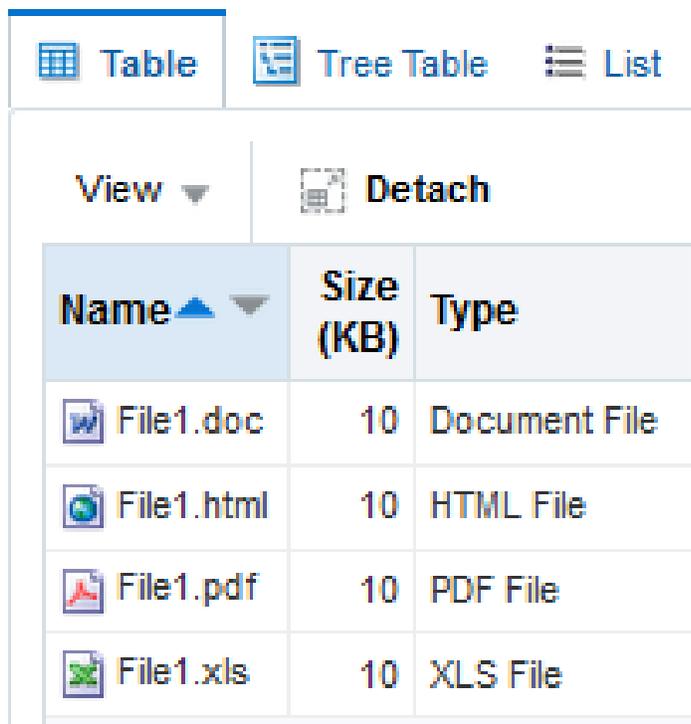
By default, the width of a tab is determined by the text displayed as the label. You can configure the tabs so that instead, the size of the tab is a certain minimum or maximum width. In cases where the text will not fit, you can set an ellipsis to display after the truncated text.

You can configure a `panelTabbed` component so that the individual tabs can be removed (closed). You can have it so that all tabs can be removed, all but the last tab can be removed, or no tabs can be removed.

You can configure when the `showDetailItem` components that contain the contents for each of the tabs will be created. When you have a small number of tabs, you can have all the `showDetailItem` components created when the `panelTabbed` component is first created, regardless of which tab is currently displayed. However, if the `panelTabbed` component contains a large number of `showDetailItem` components, the page might be slow to render. To enhance performance, you can instead configure the `panelTabbed` component to create a `showDetailItem` component only when its corresponding tab is selected. You can further configure the delivery method to either destroy a `showDetailItem` once the user selects a different tab, or to keep any selected `showDetailItem` components in the component tree so that they do not need to be recreated each time they are accessed.

The File Explorer application uses the `panelTabbed` component to display the contents in the main panel, as shown in [Figure 9-55](#).

Figure 9-55 `panelTabbed` Panes



 **Tip:**

If you want the tabs to be used in conjunction with navigational hierarchy, for example, each tab is a different page or region that contains another set of navigation items, you may want to use a navigation panel component to create a navigational menu. See [Using Navigation Items for a Page Hierarchy](#).

The `panelTabbed` component also provides overflow support for when all tabs cannot be displayed. How the overflow is handled depends on how you configure the `-tr-layout-type` skinning key. See [What You May Need to Know About Skinning and the panelTabbed Component](#).

 **Note:**

Overflow is only supported when the position attribute is set to `above`, `below`, or `both`.

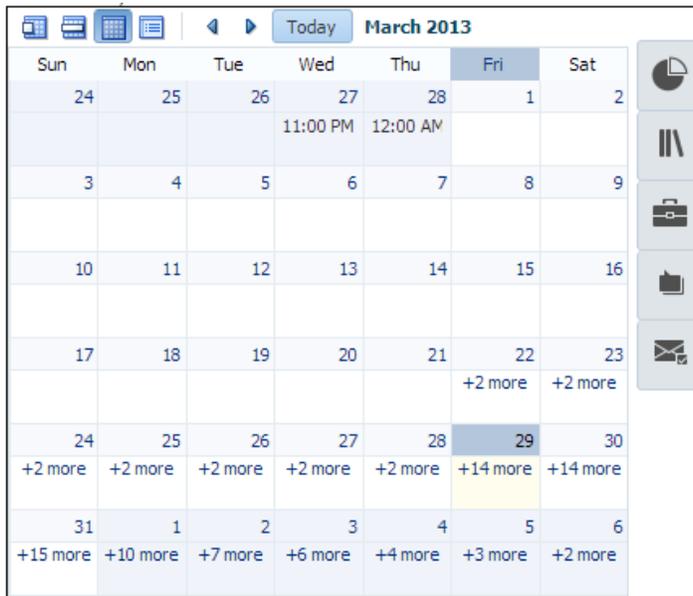
 **Performance Tip:**

The number of child components within a `panelTabbed` component, and the complexity of the child components, will affect the performance of the overflow. Set the size of the panel components to avoid overflow when possible.

To use the `panelTabbed` component, see [How to Use the panelTabbed Component](#).

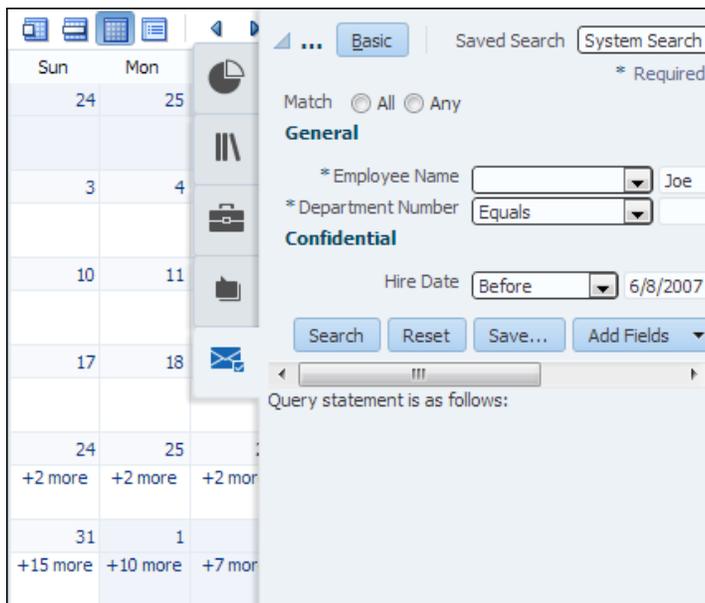
The `panelDrawer` component renders tabs attached to the side of a container component. By default, the drawer aligns to the parent of the `panelDrawer`, but you can choose another close ancestor. It can align to either the start or end of the associated component. When the user clicks a tab, the drawer opens and the content of the child `showDetailItem` becomes visible. [Figure 9-56](#) shows the `panelDrawer` with the drawers closed.

Figure 9-56 panelDrawer Component with Drawers Closed



When the user clicks one of the tabs, the associated drawer opens, as shown in [Figure 9-57](#).

Figure 9-57 panelDrawer Component with the Last Drawer Opened

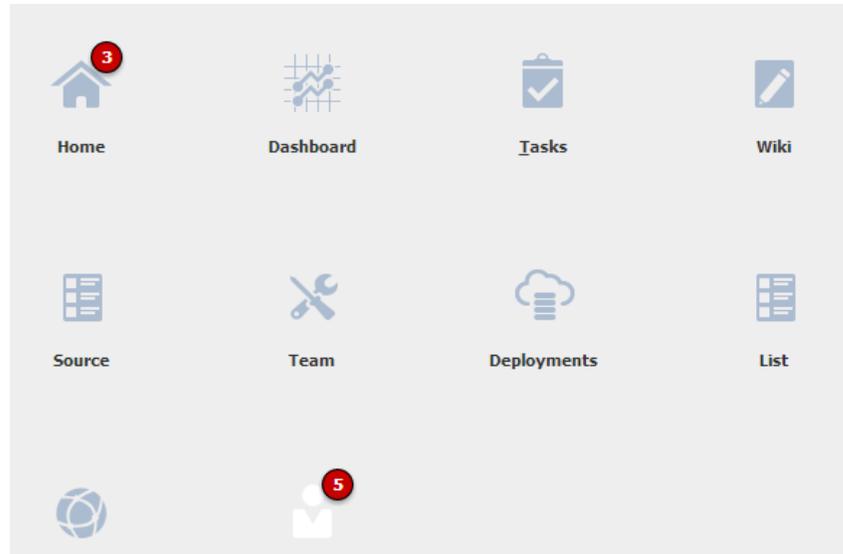


How wide the drawer opens depends on how you set the `width` attribute. If there is no value for the `width` attribute, the size of the open drawer is determined by the content contained in child the `showDetailItem` component. Otherwise, you can set the `width` attribute to a percentage of the component the `panelDrawer` is aligned to.

The `panelSpringboard` component represents its contents as a set of icons that display in either a grid fashion or in a strip. When you click on an icon, the child `showDetailItem` component associated with the clicked icon displays its contents below the strip.

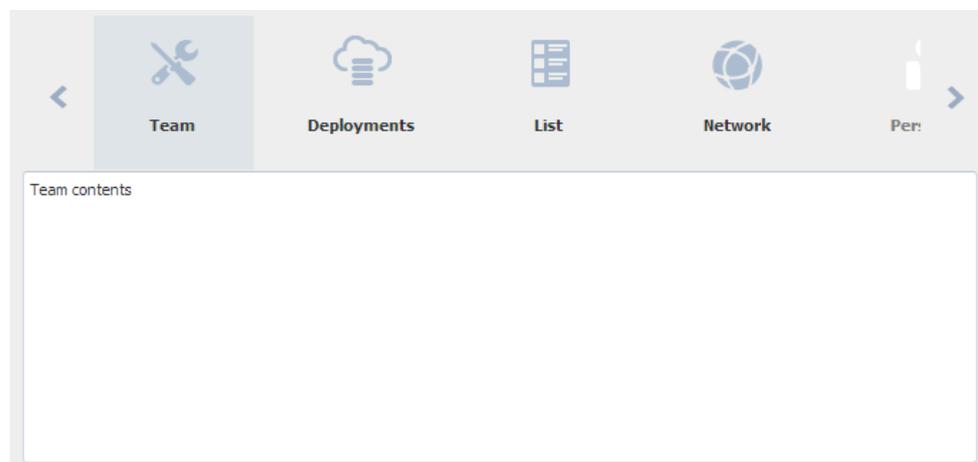
For example, [Figure 9-58](#) shows a `panelSpringboard` component that contains 10 child `showDetailItem` components, configured to display the associated icons in a grid.

Figure 9-58 `panelSpringboard` Component in Grid Mode



[Figure 9-59](#) shows the same `panelSpringboard` component after clicking the **Team** icon. The `panelSpringboard` icons move to the top, into a strip, and the content associated with the selected icon is displayed.

Figure 9-59 `panelSpringboard` Component in Strip Mode



 **Tip:**

In strip view, you can navigate the icon strip using the provided arrow buttons. In a mobile application, swiping across the content box navigates to the next (left swipe) or previous (right swipe) icon's content.

Like the `panelSpringboard` component, the `panelAccordion`, `panelTabbed`, and `panelDrawer` components use a `showDetailItem` component to provide the contents for each panel. For example, if you want to use four panes, insert four `showDetailItem` components inside the `panelAccordion`, `panelTabbed`, or `panelDrawer` components, respectively. To use the `showDetailItem` component, see [How to Use the `showDetailItem` Component to Display Content](#). You can add a toolbar to the `toolbar` facet of the `showDetailItem` component, and the toolbar will be shown whenever the `showDetailItem` is disclosed. [Figure 9-55](#) shows the toolbar used by the `showDetailItem` component in the File Explorer application.

The child `showDetailItem` component can also display a badge, used to denote some type of information about that item. For example, in the `panelSpringboard` shown in [Figure 9-58](#), badges are used to display a number of items for the Home `showDetailItem`.

For each of these components except the `panelDrawer`, you can configure when the child `showDetailItem` components will be created. When you have a small number of `showDetailItem` components, you can have all of them created when the parent component is first created. However, if the parent component contains a large number of `showDetailItem` components, the page might be slow to render. To enhance performance, you can instead configure the parent component to create a `showDetailItem` component only when it is disclosed (for example, when a tab is selected). You can further configure the delivery method to either destroy a `showDetailItem` once the user discloses a different one, or to keep any selected `showDetailItem` component in the component tree so that they do not need to be recreated each time they are accessed.

The `panelAccordion` and `panelTabbed` components can be configured to be stretched, or they can be configured to instead take their dimensions from the currently disclosed `showDetailItem` child. The `panelSpringboard` component will stretch if the parent component allows stretching of its child. If the parent does not stretch its children then the size of the `panelSpringboard` component will be based on the contents of its child `showDetailItem` component. The `panelDrawer` component will open to the size of its contained components, unless a specific width is set.

When you configure a panel component to stretch, then you can also configure the `showDetailItem` component to stretch a single child as long as it is the only child of the `showDetailItem` component.

How to Use the `panelAccordion` Component

You can use more than one `panelAccordion` component in a page, typically in different areas of the page, or nested. After adding the `panelAccordion` component, insert a series of `showDetailItem` components to provide the panes, using one `showDetailItem` for one panel. Then insert components into each `showDetailItem` to provide the panel contents. For procedures on using the `showDetailItem` component, see [How to Use the `showDetailItem` Component to Display Content](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying or Hiding Contents in Panels](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelAccordion` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Accordion** onto the JSF page.
2. In the Create Panel Accordion dialog, configure the panes for the accordion. For help with the dialog, click **Help** or press F1.
3. In the Properties window, expand the Common section.
4. Set the **DiscloseNone** to `true` if you want users to be able to collapse all panes.

By default, the value is `false`. This means one panel must remain expanded at any time.

5. If you want users to be able to rearrange the panes by dragging and dropping, expand the Behavior section, and set **Reorder** to `enabled`. The default is `disabled`.

 **Note:**

If the `panelAccordion` has components other than `showDetailItem` components (see the tip in Step 8), those components can be reordered on the client only. Therefore, any new order will not be preserved.

6. By default, the `panelAccordion` component stretches to fill available browser space. If instead, you want to use the `panelAccordion` component as a child to a component that does not stretch its children, then you need to change how the `panelAccordion` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

Set **DimensionsFrom** to one of the following:

- **children**: the `panelAccordion` component will get its dimensions from the currently disclosed `showDetailItem` component.

 **Note:**

If you use this setting, you cannot set the height of the `panelAccordion` component (for example through the `inlineStyle` or `styleClass` attributes). Doing so would cause conflict between the `panelAccordion` height and the child component height.

Similarly, you cannot set the `stretchChildren`, `flex`, and `inflexibleHeight` attributes on any `showDetailItem` component, as those settings would result in a circular reference back to the `panelAccordion` to determine size.

- **parent**: the size of the `panelAccordion` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
- **auto** (default): If the parent component to the `panelAccordion` component allows stretching of its child, then the `panelAccordion` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelAccordion` component will be based on the size of its child component.

 **Note:**

If you want the `panelAccordion` to stretch, and you also want the `showDetailItem` to stretch its contents, then you must configure the `showDetailItem` in a certain way. For details, see [How to Use the showDetailItem Component to Display Content](#).

7. You can configure when the child `showDetailItem` components will be created using the `childCreation` attribute. To do so, expand the **Other** section, and set **ChildCreation** to one of the following:
 - **immediate**: All child components are created when the `panelAccordion` component is created.
 - **lazy**: The child components are created only when they are disclosed. Once disclosed and the associated child components are rendered, they remain created in the component tree.
 - **lazyUncached**: The child components are created only when they are disclosed. Once the components are hidden, they are destroyed.
8. JDeveloper added the panes you configured in the Create Panel Accordion dialog by adding the corresponding `showDetailItem` components as a children to the

`panelAccordion` component. To add more panes, insert the `showDetailItem` component inside the `panelAccordion` component. You can add as many panes as you wish.

 **Tip:**

Accordion panels also allow you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the accordion panel.

To add contents for display in a panel, insert the desired child components into each `showDetailItem` component. For procedures, see [How to Use the `showDetailItem` Component to Display Content](#) .

How to Use the `panelTabbed` Component

Using the `panelTabbed` component to create tabbed panes is similar to using the `panelAccordion` component to create accordion panes. After adding a `panelTabbed` component, you insert a series of `showDetailItem` components to provide the tabbed panel contents for display.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying or Hiding Contents in Panels](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelTabbed` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Tabbed** onto the JSF page.
2. In the Create Panel Tabbed dialog, configure the panes for the panel. For help with the dialog, click **Help** or press F1.
3. In the Properties window, expand the Common section.
4. If you want users to be able to close (remove) tabs, then set **TabRemoval**. You can set it to allow all tabs to be removed, or all but the last tab. You must implement a handler to do the actual removal and configure the listeners for the associated `showDetailItem` components. You can override this on an individual `showDetailItem` component, so that an individual tab cannot be removed (a close icon does not display), or so that the closed icon is disabled.

When tabs are configured to be removed, a close icon is displayed at the end of the tab (whether it was disclosed through clicking or by tabbing through the tabs).

See [How to Use the `showDetailItem` Component to Display Content](#) .

 **Note:**

Tab removal is only supported when the position attribute is set to `above`, `below`, or `both`.

- By default, the size of the tabs is determined by the length of the text used as the label. You can instead set the tabs to be a certain size, and then have any text that does not fit display as truncated text with an ellipsis. To do so, set the following:
 - `maxTabSize`: Set to a size in pixels. The tabs will never be larger than this size. To fill all available tab space, set to `infinity`. This is the default.
 - `minTabSize`: Set to a size in pixels. The tabs will never be smaller than this size.
 - `truncationStyle`: Set to `ellipsis` if you want an ellipsis to display after truncated text that cannot fit, based on the `maxTabSize`. If set to `none`, then if the text does not fit on the tab, it will simply be truncated. Note that if you do not set `maxTabSize`, then the tab will always be as large as the text needs.

 **Note:**

Truncation and expansion are only supported when you set `truncationStyle` to `ellipsis`. If set to `none`, then `maxTabSize` and `minTabSize` are ignored, and the size of the tab is based on the length of the text.

- By default, the `panelTabbed` component stretches to fill available browser space. If instead, you want to use the `panelTabbed` component as a child to a component that does not stretch its children, then you need to change how the `panelTabbed` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

Set **DimensionsFrom** to one of the following:

- `disclosedChild`: the `panelTabbed` component will get its dimensions from the currently disclosed `showDetailItem` component.

 **Note:**

If you use this setting, you cannot set the height of the `panelTabbed` component (for example through the `inlineStyle` or `styleClass` attributes). Doing so would cause conflict between the `panelTabbed` height and the child component height.

- `parent`: the size of the `panelTabbed` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
 - `auto` (default): If the parent component to the `panelTabbed` component allows stretching of its child, then the `panelTabbed` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `panelTabbed` component will be based on the size of its child component.
7. You can configure when the child `showDetailItem` components will be created using the `childCreation` attribute. To do so, expand the **Other** section, and set **ChildCreation** to one of the following:
- `immediate`: All child components are created when the `panelTabbed` component is created.
 - `lazy`: The child components are created only when they are disclosed. Once disclosed and the associated child components are rendered, they remain created in the component tree.
 - `lazyUncached`: The child components are created only when they are disclosed. Once the components are hidden, they are destroyed.
8. JDeveloper created the panes you configured in the Create Panel Tabbed dialog by adding the corresponding `showDetailItem` components as a children to the `panelTabbed` component. To add more panes, insert the `showDetailItem` component inside the `panelTabbed` component. You can add as many panes as you wish.

 **Tip:**

The `panelTabbed` component also allow you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the tabbed panel.

To add contents for display in a tab, insert the desired child components into each `showDetailItem` component. For procedures, see [How to Use the showDetailItem Component to Display Content](#).

How to Use the panelDrawer Component

Using the `panelDrawer` component to create tabbed panes is similar to using the `panelTabbed` component to create tabbed panes. After adding a `panelDrawer` component, you insert a series of `showDetailItem` components to provide the drawer contents for display. You can also control the closing behavior of the panel drawer.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying or Hiding Contents in Panels](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelDrawer` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Drawer** onto the JSF page.
2. In the Create Panel Drawer dialog, configure the panes for the panel. For help with the dialog, click **Help** or press F1.
3. In the Properties window, expand the Common section.
4. Set **AlignId** to the component to which the `panelDrawer` should align. Click the icon that appears when you hover over the property field, and choose **Edit** to open the Edit Property: AlignId dialog and choose the component. If you do not set the `alignId` attribute, the `panelDrawer` will align to its parent.

 **Note:**

The `panelDrawer` does not support overflow content. Therefore, the component to which the `panelDrawer` is aligned must be tall enough to accommodate all the tabs and their contents.

5. Set the **width** and **height** of the drawer. By default, the `panelDrawer` component stretches to the size of the contents of the `showDetailItem` component. In turn, the `showDetailItem` will allow stretching if the following is true:
 - The `panelDrawer` has `width` and `height` attributes defined.
 - The `showDetailItem` contains a single child.
 - The child component of the `showDetailItem` has no value set for the `width`, `height`, `margin`, `border`, or `padding`.
 - The child must be capable of being stretched.

 **Note:**

If the size of the content will change after the drawer is open (for example you toggle a `showDetail` inside the drawer which exposes new content), you should set the `width` and `height` attributes to the largest expected size. Otherwise, the resized content may not display properly.

6. Set **MaximumHeight** and **MaximumWidth** as needed. By default, it is set to 100%.
7. To override the closing behavior of the `panelDrawer` component, set the **AutoDismiss** option. You can use this option when you turn on the page composer and use the page composer tool to edit the items in the `panelDrawer`. By default, the **AutoDismiss** option is set to `auto`, which is as per the skin property of `af|panelDrawer-tr-auto-dismiss` value. You can set one of the following values to **AutoDismiss** option in the **Other** section of the Property Inspector:
 - **auto**: Displays the `panelDrawer` according to the skin property `af|panelDrawer-tr-auto-dismiss` value.
 - **focusLoss**: Dismisses the `panelDrawer` when you click outside of the component and overrides skin property `af|panelDrawer-tr-auto-dismiss` value.
 - **none**: Keeps the `panelDrawer` open even when you click outside of the component and overrides skin property `af|panelDrawer-tr-auto-dismiss` value.
8. Expand the Appearance section and set **ShowHandles**. By default, it is set to `always`, which means the handles will always display. You can also set it to `whenOpen`, which will only show the handle when the drawer is open. You will need to programmatically open the drawer by setting the `disclosed` attribute on the corresponding `showDetailItem` to `true`. For example, you may want to use buttons to open the drawers, instead of the handles. The action associated with the button would set a `showDetailItem`'s `disclosed` attribute to `true`.
9. JDeveloper created the panes you configured in the Create Panel Drawer dialog by adding the corresponding `showDetailItem` components as a children to the `panelDrawer` component. To add more panes, insert the `showDetailItem` component inside the `panelDrawer` component. You can add as many panes as you wish.

 **Tip:**

Panel Drawers also allow you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the panel drawer.

To add contents for display in a drawer, insert the desired child components into each `showDetailItem` component. For procedures, see [How to Use the showDetailItem Component to Display Content](#).

How to Use the panelSpringboard Component

The `panelSpringboard` contains a series of `showDetailItem` components, similar to the other panel components. Each `showDetailItem` is represented by an icon. You insert components into each `showDetailItem` to provide the panel contents. The `panelSpringboard` supports swiping gestures in mobile applications. For procedures on using the `showDetailItem` component, see [How to Use the showDetailItem Component to Display Content](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying or Hiding Contents in Panels](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelSpringboard` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Springboard** onto the JSF page.
2. In the Properties window, expand the Appearance section, and set **DisplayMode** to determine how the `panelSpringboard` should display when it is first rendered. Set it to `grid` to display only the icons, as shown in [Figure 9-58](#). Set it to `strip` to display the icons along the top and the contents of the selected icon below the strip, as shown in [Figure 9-59](#).

Tip:

If you want to be able to switch between the two modes, you need to add JavaScript to your page. See [What You May Need to Know About Switching Between Grid and Strip Mode](#).

3. If you want some logic to execute when the display mode changes, expand the Advanced section and set **SpringboardChangeListener** to a method on a bean that will handle this logic.
4. At runtime, by default, all child `showDetailItem` components are created when the `panelSpringboard` component is created. If there will be a large number of children, to improve performance you can configure the `panelSpringboard` either so that it creates the child `showDetailItem` component only when the tab is selected, or so that it creates the child `showDetailItem` component only when it's selected the first time, and from that point on it remains created.

You configure when the child components will be created using the `childCreation` attribute. To do so, expand the **Behavior** section, and set **ChildCreation** to one of the following:

- `immediate`: All `showDetailItem` components are created when the `panelSpringboard` component is created.
- `lazy`: The `showDetailItem` component is created only when the associated icon is selected. Once an icon is selected and the associated `showDetailItem` is rendered, the `showDetailItem` component remains created in the component tree.

- `lazyUncached`: The `showDetailItem` component is created only when the associated icon is selected. Once another icon is selected, the `showDetailItem` component is destroyed.
5. Insert the `showDetailItem` components inside the `panelSpringboard` component. You can add as many as you wish. The order in which you add them as children will be the order in which they display in the springboard.

 **Tip:**

The `panelSpringboard` component also allows you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the `panelSpringboard`.

To add contents for display, insert the desired child components into each `showDetailItem` component. For procedures, see [How to Use the `showDetailItem` Component to Display Content](#) .

What You May Need to Know About Switching Between Grid and Strip Mode

By default, the `panelSpringboard` renders the first time in grid mode. When a user clicks an icon, the `panelSpringboard` fires a `SpringboardChangeListener` event and changes to strip mode. If you want to be able to switch between the two modes, you need to listen for that event, determine the source (the `panelSpringboard`), and set the `displayMode` attribute to the desired mode.

For example, to set the display mode to grid, you might use the JavaScript shown in the following example.

```
<af:resource type="javascript">
function backToGrid(actionEvent)
{
    actionEvent.cancel();
    var eventSource = actionEvent.getSource();
    var object_navigator = eventSource.findComponent("panelSpringboardId");
    object_navigator.setProperty(AdfRichPanelSpringboard.DISPLAY_MODE, "grid",
        true);
}
</af:resource>
```

You might then call that code from a link:

```
<af:link id="logo" text="Back to Grid">
    <af:clientListener type="click" method="backToGrid"/>
</af:link>
```

For information about using JavaScript on a page, see [Using ADF Faces Client-Side Architecture](#) .

How to Use the `showDetailItem` Component to Display Content

The `showDetailItem` components are used as child components to the `panelAccordion`, `panelTabbed`, `panelDrawer`, or `panelSpringboard` component only.

Each `showDetailItem` component corresponds to one panel. Dialogs for the `panelAccordion`, `panelTabbed`, and `panelDrawer` components create the `showDetailItem` components for you. You need to add the `showDetailItem` components to the `panelSpringboard` component manually. You can also add more `showDetailItem` components into the other parent components as needed. You then insert the child components for display into the `showDetailItem` components.

The `disclosed` attribute on a `showDetailItem` component specifies whether to show (disclose) or hide (undisclose) the corresponding panel contents. When the `disclosed` attribute is `false`, the contents are hidden (undisclosed). When the attribute is set to `true`, the contents are shown (disclosed). You do not have to write any code to enable the toggling of contents from disclosed to undisclosed, and vice versa. ADF Faces handles the toggling automatically.

The following procedure assumes you have already added a `panelAccordion`, `panelTabbed`, `panelDrawer`, or `panelSpringboard` component to the JSF page, as described in [How to Use the panelAccordion Component](#), [How to Use the panelTabbed Component](#), [How to Use the panelDrawer Component](#), and [How to Use the panelSpringboard Component](#), respectively.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying or Hiding Contents in Panels](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To add panel contents using a `showDetailItem` component:

1. Insert one or more `showDetailItem` components inside the parent component, such as `panelAccordion`, by dragging and dropping a **Show Detail Item** component from Layout panel of the Components window.
2. In the Properties window, expand the **Appearance** section.
3. Set **Text** to the label you want to display for this panel, tab, or icon. If you used a dialog to create the panels, this will already be set for you.
4. To add an icon, set **Icon** to the URI of the image file to use. If you used a dialog to create the panels, this will already be set for you. You can also set **HoverIcon**, **DepressedIcon**, **DisabledIcon**.

 **Note:**

Because alternative text cannot be provided for this icon, in order to create an accessible interface, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

5. If the `showDetailItem` component is being used inside a `panelAccordion` component configured to stretch, you can configure the `showDetailItem` to stretch and in turn stretch its contents, however, the `showDetailItem` component must contain only one child component. You need to set **Flex** and the **StretchChildren** for each `showDetailItem` component.

Use the following attributes on each `showDetailItem` component to control the flexibility of panel contents:

- **Flex:** Specifies a nonnegative integer that determines how much space is distributed among the `showDetailItem` components of one `panelAccordion` component. By default, the value of the `flex` attribute is 0 (zero), that is, the panel contents of each `showDetailItem` component are inflexible. To enable flexible contents in a panel, specify a `flex` number larger than 0, for example, 1 or 2. A larger `flex` value means that the contents will be made larger than components with lower `flex` values. For two flexible components, their height sizes are exactly proportionate to the `flex` values assigned. If component A has `flex` set to 2 and component B has `flex` set to 1, then the height of component A is two times the height of component B.
- **InflexibleHeight:** Specifies the number of pixels a panel will use. The default is 100 pixels. This means if a panel has a `flex` value of 0 (zero), ADF Faces will use 100 pixels for that panel, and then distribute the remaining space among the nonzero panes. If the contents of a panel cannot fit within the `panelAccordion` container given the specified `inflexibleHeight` value, ADF Faces automatically moves nearby contents into overflow menus (as shown in [Figure 9-53](#)). Also, if a panel has a nonzero `flex` value, this will be the minimum height that the panel will shrink to before causing other panes to be moved into the overflow menus.
- **StretchChildren:** When set to `first`, stretches a single child component. However, the child component must allow stretching. See [What You May Need to Know About Geometry Management and the showDetailItem Component](#).

For example, the File Explorer application uses `showDetailItem` components to display contents in the navigator panel. Because the Search Navigator requires more space when both navigators are expanded, its `flex` attribute is set to 2 and the `showDetailItem` component for the Folders Navigator uses the default `flex` value of 1. This setting causes the Search Navigator to be larger than the Folders Navigator when it is expanded.

 **Note:**

Instead of directly setting the value for the `flex` attribute, the File Explorer application uses an EL expression that resolves to a method used to determine the value. Using an EL expression allows you to programmatically change the value if you decide at a later point to use metadata to provide model information.

The user can change the panel heights at runtime, thereby changing the value of the `flex` and `inflexibleHeight` attributes. Those values can be persisted so that they remain for the duration of the user's session. For information, see [Allowing User Customization on JSF Pages](#).

Note the following additional information about flexible accordion panel contents:

- There must be two or more panes (`showDetailItem` components) with `flex` values larger than 0 before ADF Faces can enable flexible contents. This is because ADF Faces uses the `flex` ratio between two components to determine how much space to allocate among the panel contents. At runtime,

two or more panes must be expanded before the effect of flexible contents can be seen.

- If the `showDetailItem` component has only one child component and the `flex` value is nonzero, and the `stretchChildren` attribute is set to `first`, ADF Faces will stretch that child component regardless of the `discloseMany` attribute value on the `panelAccordion` component.
- When all `showDetailItem` components have `flex` values of 0 (zero) and their panel contents are disclosed, even though the disclosed contents are set to be inflexible, ADF Faces will stretch the contents of the last disclosed `showDetailItem` component as if the component had a `flex` value of 1, but only when that `showDetailItem` component has one child only, and the `stretchChildren` attribute is set to `first`. If the last disclosed panel has more than one child component or the `stretchChildren` attribute is set to `none`, the contents will not be stretched.

Even with the `flex` attribute set, there are some limitations regarding geometry management. See [What You May Need to Know About Geometry Management and the `showDetailItem` Component](#).

6. If the `showDetailItem` component is being used inside a `panelSpringboard` component configured to stretch, you can configure the `showDetailItem` to stretch and in turn stretch its contents. However, the `showDetailItem` component must contain only one child component, and its child must not have any width, height, margin, border, or padding set on it.

To stretch the contents, set **StretchChildren** to `first` for each `showDetailItem` component.

 **Tip:**

If the component that will be placed in the `showDetailItem` does not support stretching, or if you need to place more than one component as a child to the `showDetailItem`, then you must set `stretchChildren` to `none`, as stretching will not be supported.

7. Expand the Behavior section. Set **DisclosureListener** to the `disclosureListener` method in a backing bean you want to execute when this panel, tab, or icon is selected by the user.

For information about server disclosure events and event listeners, see [What You May Need to Know About Disclosure Events](#).

8. Set **Disabled** to `true` if you want to disable this panel, tab, or icon (that is, the user will not be able to select the panel or tab).
9. Set **Disclosed** to `true` if you want this panel, tab, or icon to show its child components. If you created a `panelAccordion` component using a dialog and selected **Disclosed**, then this has already been set for you.

When you add a `showDetailItem` manually, the `disclosed` attribute is set to `false`. This means the contents for this panel, tab, or icon are hidden.

 **Note:**

There is difference between the `disclosed` and `rendered` attributes. If the `rendered` attribute value is `false`, it means that this accordion header bar or tab link and its corresponding contents are not available at all to the user. However, if the `disclosed` attribute is set to `false`, it means that the contents of the item are not currently visible, but may be made visible by the user because the accordion header bar or tab link are still visible.

If none of the `showDetailItem` components has the `disclosed` attribute set to `true`, ADF Faces automatically shows the contents of the first enabled `showDetailItem` component (except when it is a child of a `panelAccordion` component, which has a setting for zero disclosed panes).

 **Note:**

While the user can change the value of the `disclosed` attribute by displaying or hiding the contents, the value will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).

10. For `showDetailItem` components used in a `panelAccordion` component, expand the Behavior section, and set **DisplayIndex** to reflect the order in which the `showDetailItem` components should appear. If you simply want them to appear in the order in which they are in the page's code, then leave the default, `-1`.

 **Tip:**

If some `showDetailItem` components have `-1` as the value for `displayIndex`, and others have a positive number, those with the `-1` value will display after those with a positive number, in the order they appear in the page's code.

 **Tip:**

This value can be changed at runtime if the parent `panelAccordion` component is configured to allow reordering.

11. If you chose to allow tab removal for a `panelTabbed` component, expand the Behavior section and set **Remove** to one of the following:
 - **inherit**: The corresponding tab can be removed if the parent `panelTabbed` component is configured to allow it. This is the default.
 - **no**: The corresponding tab cannot be removed, and will not display a close icon.

- **disabled:** The corresponding tab will display a disabled close icon.

Set `ItemListener` to an EL expression that resolves to a handler method that will handle the actual removal of a component.

12. To add toolbar buttons to a panel in a `panelAccordion` component, in the Components window, from the Layout panel, in the Menus and Toolbar Containers group, insert a **Toolbar** into the `toolbar` facet of the `showDetailItem` component that defines that panel (if you selected **Toolbar** in the Create Panel Accordion dialog, then the `toolbar` component has already been added for you). Then, insert the desired number of button components into the `toolbar` component. Although the `toolbar` facet is on the `showDetailItem` component, it is the `panelAccordion` component that renders the toolbar and its buttons. For information about using `toolbar` and `button` components, see [Using Toolbars](#).

You can configure the toolbar display by expanding the Other section and setting the **ToolbarVisibility** property to one of the following:

- **auto:** Displays the toolbar either always or disclosed depending on the internal implementation of `panelAccordion`. The default is **auto**.
- **always:** Displays the toolbar on all detail items even if the items are in disclosed state.
- **disclosed:** Displays the toolbar only if detail items are in disclosed state.

 **Note:**

When an accordion panel is collapsed, ADF Faces does not display the toolbar and its buttons. The toolbar and its buttons are displayed in the panel header only when the panel is expanded.

13. To add additional information about a `showDetailItem` (for example, for a `panelSpringboard` you may want to display notifications), enter a value for **Badge**.
14. To add contents to the panel, insert the desired child components into each `showDetailItem` component.

What You May Need to Know About Geometry Management and the `showDetailItem` Component

The `panelAccordion`, `panelTabbed`, `panelDrawer` and `panelSpringboard` components can be configured to stretch when they are placed inside a component that uses geometry management to stretch its child components. However, by default, the `showDetailItem` will not stretch its children.

For the `panelAccordion` component, the `showDetailItem` component will stretch only if the `discloseMany` attribute on the `panelAccordion` component is set to `true` (that is, when multiple panes may be expanded to show their inflexible or flexible contents), the `showDetailItem` component contains only one child component, and the `showDetailItem` component's `stretchChildren` attribute is set to `first`.

For the other panel components, the `showDetailItem` component will allow stretching if:

- It contains only a single child

- Its `stretchChildren` attribute is set to `first`
- The child has no width, height, border, and padding set
- The child must be capable of being stretched

When all of the preceding bullet points are true, the `showDetailItem` component can stretch its child component. The following components can be stretched inside the `showDetailItem` component:

- `decorativeBox` (when configured to stretch)
- `deck`
- `calendar`
- `inputText` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection` (when configured to stretch)
- `panelDashboard` (when configured to stretch)
- `panelGroupLayout` (only when the `layout` attribute is set to `scroll` or `vertical`)
- `panelLabelAndMessage` (when configured to stretch)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a `showDetailItem` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only when the `layout` attribute is set to `default` or `horizontal`)
- `panelHeader`
- `panelList`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch as a child to a component that stretches its child components. Therefore, if you need to place one of the components that cannot be stretched as a child of a `showDetailItem` component, you need to wrap that component in different component that does not stretch its child components.

For example, if you want to place content in a `panelList` component and have it be displayed in a `showDetailItem` component, you might place a `panelGroupLayout` component with its `layout` attribute set to `scroll` as the child of the `showDetailItem`

component, and then place the `panelList` component in that component. See [Geometry Management and Component Stretching](#).

What You May Need to Know About `showDetailItem` Disclosure Events

The `showDetailItem` component inside of panel components supports queuing of disclosure events so that validation is properly handled on the server and on the client.

In general, for any component with the `disclosed` attribute, by default, the event root for the client `AdfDisclosureEvent` is set to the event source component: only the event for the panel whose `disclosed` attribute is `true` gets sent to the server. However, for the `showDetailItem` component that is used inside of `panelAccordion`, `panelTabbed`, or `panelDrawer` component, the event root is that panel component (that is, the event source parent component, not the event source component). This ensures that values from the previously disclosed panel will not get sent to the server.

For example, suppose you have two `showDetailItem` components inside a `panelTabbed` component with the `discloseMany` attribute set to `false` and the `discloseNone` attribute set to `false`. Suppose the `showDetailItem 1` component is disclosed but not `showDetailItem 2`. Given this scenario, the following occurs:

- On the client:
 - When a user clicks to disclose `showDetailItem 2`, a client-only disclosure event gets fired to set the `disclosed` attribute to `false` for the `showDetailItem 1` component. If this first event is not canceled, another client disclosure event gets fired to set the `disclosed` attribute to `true` for the `showDetailItem 2` component. If this second event is not canceled, the event gets sent to the server; otherwise, there are no more disclosure changes.
- On the server:
 - The server disclosure event is fired to set the `disclosed` attribute to `true` on the `showDetailItem 2` component. If this first server event is not canceled, another server disclosure event gets fired to set the `disclosed` attribute to `false` for the `showDetailItem 1` component. If neither server event is canceled, the new states get rendered, and the user will see the newly disclosed states on the client; otherwise, the client looks the same as it did before.

For the `panelAccordion` component with the `discloseMany` attribute set to `false` and the `discloseNone` attribute set to `true`, the preceding information is the same only when the disclosure change forces a paired change (that is, when two disclosed states are involved). If only one disclosure change is involved, there will just be one client and one server disclosure event.

For the `panelAccordion` component with the `discloseMany` attribute set to `true` (and any `discloseNone` setting), only one disclosure change is involved; there will just be one client and one server disclosure event.

For additional information about disclosure events, see [What You May Need to Know About Disclosure Events](#).

What You May Need to Know About Skinning and the `panelTabbed` Component

You can use the `-tr-layout-type` skinning key to configure how the `panelTabbed` component handles overflow when its parent container is too small to display all the tabs. This horizontal compressed layout can display either overflow button(s) or can roll to show hidden tabs, similar to a conveyor belt.

If the `panelTabbed` component is vertically compressed, then the vertical compressed layout can roll to show the hidden conveyor belt icons. The expand/collapse icon that is built over the conveyor belt can be used to either view icons only or to view icons with text labels.

You can configure `panelTabbed` components to display the expand/collapse icon in the `web.xml` file using the `oracle.adf.view.rich.PANEL_TABBED_DEFAULT_VERTICAL_TAB_MODE` property. The default value for this configuration property is `false`. To display the expand/collapse icon for `panelTabbed` components at application level, set the `oracle.adf.view.rich.PANEL_TABBED_DEFAULT_VERTICAL_TAB_MODE` configuration property to `true`.

For a vertical compressed layout, ADF provides a `verticalTabMode` attribute for the `panelTabbed` component that determines whether to display the conveyor belt icons only or to display the conveyor belt icons with text labels. You can set the `verticalTabMode` attribute to the following values:

- `iconOnly`: By default, the value of this attribute is set to `iconOnly`, which displays only the conveyor belt icons.
- `iconAndText`: Displays the conveyor belt icons with text labels.
- `auto`: Automatically displays the conveyor belt icons with text labels depending on the browser resolution.

When the `verticalTabMode` attribute is set to `auto`, the conveyor belt icons with text labels are displayed when the browser resolution exceeds the set viewport threshold value. The viewport threshold value can be configured from the skin. The default value of the viewport threshold size is 768px. The `resizeNotify` method of the `panelTabbed` component is configured such that when the `verticalTabMode` attribute is set to `auto`, `af_iconified` class is set to the root element of the `panelTabbed` component when viewport size is less than the threshold value. If the viewport size exceeds the threshold value, then the `resizeNotify` method is configured to remove the `af_iconified` class that was set to the root element of the `panelTabbed` component.

Note:

When the user clicks the expand/collapse icon, the `auto` behavior is disabled until the session expires or until the user moves out of the corresponding page. This behavior persists on any PPR request within the page and overrides other settings.

To display the conveyor belt icons with text labels by default, set the `oracle.adf.view.rich.PANEL_TABBED_DEFAULT_VERTICAL_TAB_MODE` configuration property to `true`. This will override the default value of `verticalTabMode` attribute.

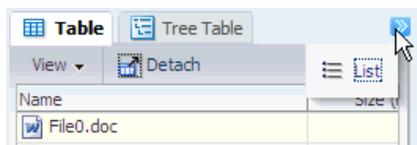


Note:

Overflow is only supported when the `position` attribute is set to `above`, `below`, or `both`.

Figure 9-60 shows the overflow compressed layout. When the user clicks the overflow icon, a popup displays showing the items that are hidden.

Figure 9-60 Overflow Compressed Layout

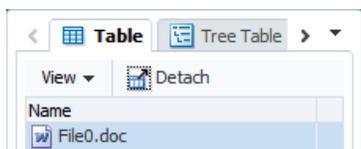


The following example shows how you use the skinning key to configure an overflow layout.

```
af|panelTabbed {
  -tr-layout-type: overflow;
}
```

Figure 9-61 shows the horizontal conveyor compressed layout. When the user clicks the overflow icon, the tabs that were hidden slide into place, similar to a conveyor belt. Accordingly, tabs on the other end are hidden.

Figure 9-61 Conveyor Belt Compressed Layout



The following example shows how you can use the skinning key to use a conveyor belt layout.

```
af|panelTabbed {
  -tr-layout-type: conveyor;
}
```

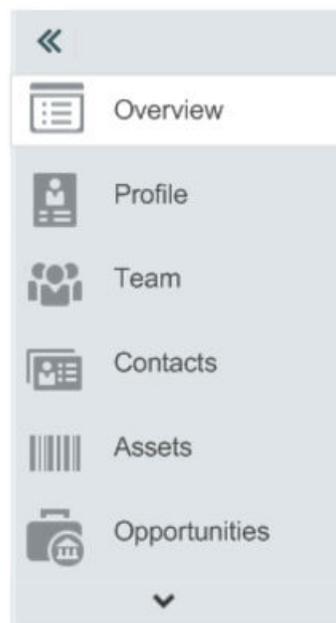
Figure 9-62 shows the vertical conveyor compressed layout with the expand icon at the top. When the user clicks the overflow icon, the icons that were hidden slide into place, similar to a conveyor belt. Accordingly, icons on the other end are hidden. When the user clicks the expand icon, icons with text labels are displayed.

Figure 9-62 Conveyor Belt Vertical Compressed Layout



Figure 9-63 shows the vertical conveyor compressed layout with icons and text labels and the collapse icon at the top. When the user clicks the overflow icon, the icons that were hidden slide into place, similar to a conveyor belt. Accordingly, icons on the other end are hidden. When the user clicks the collapse icon, only icons are displayed.

Figure 9-63 Conveyor Belt Vertical Compressed Layout with Text Labels



 **Note:**

In order for the `panelTabbed` component to support a compressed layout, its parent component must either stretch its children or be a set width.

Therefore, the following layout configurations are not supported:

- Using a parent container that does not stretch its children.
- Using a parent container that displays multiple children horizontally without explicit sizes for each child. For example, a `panelGroupLayout` with `layout='horizontal'` would be invalid, but `panelSplitter` is valid because it has an explicitly set splitter position.
- Setting the compressed layout component with a `styleClass` or `inlineStyle` that assigns a percentage width value. Note that this includes assigning `styleClass='AFStretchWidth'` on a compressed layout component.

For information about skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding a Transition Between Components

Using the ADF Faces deck component, you can control the transition between the content. The deck component is just a container component that controls the display of child components.

When you want to provide a slide show-like transition between content, you use the deck component as a parent container, and then place other layout components as children to the deck component. The content in the child layout components will then transition between each other. You use the `transition` tag to determine the type of transition.

 **Note:**

The deck component is simply a container component that keeps track of which child component to display. The transition tags handle the animation between the components. You will need to create the navigation controls to handle the actual transition.

For example, say you want to transition between content in two `panelGroupLayout` components. You can wrap them both in a deck component and insert two `transition` tags to handle the transition animation going forward and backward, as shown in the following example.

```
<af:deck id="deck1" displayedChild="pg10">
  <af:transition triggerType="backNavigate" transition="flipEnd"/>
  <af:transition triggerType="forwardNavigate" transition="fade"/>
  <af:panelGroupLayout id="pg1" layout="scroll">
    <af:panelBox text="PanelBox1" id="pb1" background="light">
      <f:facet name="toolbar"/>
    <af:panelGroupLayout id="pg11" layout="scroll">
      <af:outputText id="ot1" value="Card 1"/>
    </af:panelGroupLayout>
  </af:panelGroupLayout>
</af:deck>
```

```
        <af:image source="/images/icons-large/horizontalBarGraph.png"
                id="i1"/>
    </af:panelGroupLayout>
</af:panelBox>
</af:panelGroupLayout>
<af:panelGroupLayout id="pgl2" layout="scroll">
    <af:outputText id="ot2" value="Card 2"/>
</af:panelGroupLayout>
</af:deck
```

How to Use the Deck Component

The `deck` component acts as a container and handles determining which component to display. The `transition` tag determines the animation to use when going forward or backward.

To use the deck component:

1. In the Component Palette, from the Layout panel, drag and drop a **Deck** onto the page.
2. From the Operations panel, drag and drop a Transition.
3. In the Insert Transition dialog, select a **Transition** and **Transition Type**. For a description of the transitions, see the ADF Faces Tag Reference documentation.

Note:

Not all browser versions support all transitions. See the Tag Reference document for more information.

If you want both forward and backward transitions, you will need to add two `transition` tags, one for each.

4. Add layout components and their content, as children to the deck component.
5. Select the `deck` component and in the Property Inspector, for **DisplayedChild**, enter the ID for the component that you want to display first.
6. The `deck` component can stretch to fill available browser space. If instead, you want to use the `deck` component as a child to a component that does not stretch its children, then you need to change how the `deck` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

 **Note:**

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

To use the `dimensionsFrom` attribute, set **DimensionsFrom** to one of the following:

- `children`: the deck component will get its dimensions from its child components.
- `parent`: the size of the deck component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
- `auto`: If the parent component to the deck component allows stretching of its child, then the deck component will stretch to fill the parent. If the parent does not stretch its children then the size of the deck component will be based on the size of its child component.

See [What You May Need to Know About Geometry Management and the deck Component](#).

 **Note:**

When using an animation, you will not see components that use programmatic geometry management appear in their final state until after the animation is complete. This effect may be more pronounced depending on the complexity of your component structure, so you may need to evaluate whether an animation is appropriate.

7. You need to add components to handle the actual transition. For example, you might add a `poll` and `commandImageLinks`. When a `commandImageLink` is selected, the `poll` recognizes this and advances the deck. For an example, see the `deck` component in the ADF Faces Components demo application.

What You May Need to Know About Geometry Management and the deck Component

The `deck` component can stretch child components and it can also be stretched. The following components can be stretched inside the `deck` component:

- `inputText` (when configured to stretch)
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection`
- `panelDashboard`
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a facet of the `deck` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelHeader`
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `showDetailHeader`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch as a child to a component that stretches its child components. Therefore, if you need to place one of the components that cannot be stretched as a child to the `deck` component, wrap that component in a transition component that does not stretch its child components.

For example, if you want to place content in a `deck` component and have it flow, you could place a `panelGroupLayout` component with its `layout` attribute set to `scroll` in

the `deck` component, and then place child components in that `panelGroupLayout` component. See [Nesting Components Inside Components That Allow Stretching](#).

Displaying Items in a Static Box

When you want to display specific information, create hierarchical content, or want to transition to a different look and feel on a page, you can use the ADF Faces `panelHeader` or `decorativeBox` component. The `panelHeader` component displays the content, whereas the `decorativeBox` provides the transition.

You can use the `panelHeader` component when you want header type functionality, such as message display or associated help topics, but you do not have to provide the capability to show and hide content.

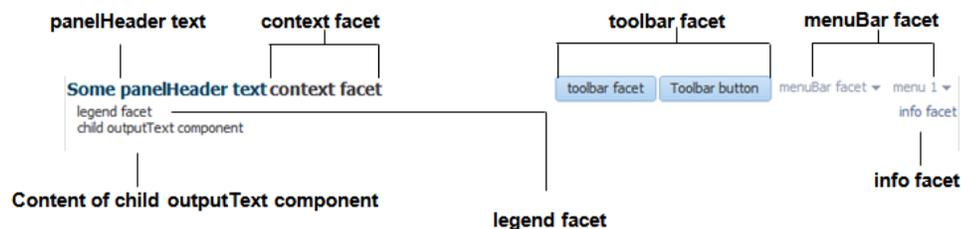
You can use the `decorativeBox` component when you need to transition to a different look and feel on the page. The `decorativeBox` component uses themes and skinning keys to control the borders and colors of its different facets. See [What You May Need to Know About Skinning and the `decorativeBox` Component](#).

The `panelHeader` component offers facets for specific types of components and the ability to open a help topic from the header. The following are the facets supported by the `panelHeader` component:

- `context`: Displays information in the header alongside the header text.
- `help`: Displays help information. Use only for backward compatibility. Use the `helpTopicId` attribute on the `panelHeader` component instead.
- `info`: Displays information beneath the header text, aligned to the right.
- `legend`: If help text is present, displays information to the left of the help content and under the `info` facet's content. If help text is not present, the legend content will be rendered directly under the header.
- `toolbar`: Displays a toolbar, before the menu bar.
- `menuBar`: Displays a menu bar, after the toolbar.

[Figure 9-64](#) shows the different facets in the `panelHeader` component.

Figure 9-64 `panelHeader` and Its Facets



When there is not enough space to display everything in all the facets of the title line, the `panelHeader` text is truncated and displays an ellipsis. When the user hovers over the truncated text, the full text is displayed in a tooltip, as shown in [Figure 9-65](#).

Figure 9-65 Text for the panelHeader Is Truncated



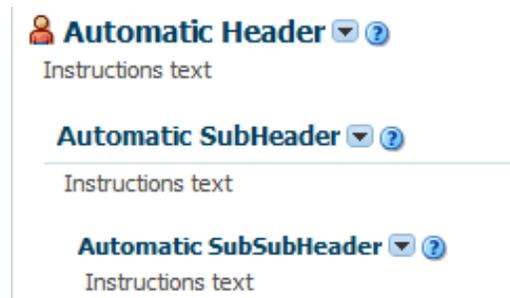
When there is more than enough room to display the contents, the extra space is placed between the `context` facet and the toolbar, as shown in [Figure 9-66](#).

Figure 9-66 Extra Space Is Added Before the Toolbar



You can configure `panelHeader` components so that they represent a hierarchy of sections. For example, as shown in [Figure 9-67](#), you can have a main header with a subheader and then a heading level 1 also with a subheader.

Figure 9-67 Creating Subsections with the panelHeader Component



Create subsections by nesting `panelHeader` components within each other. When you nest `panelHeader` components, the heading text is automatically sized according to the hierarchy, with the outermost `panelHeader` component having the largest text.

Note:

Heading sizes are determined by default by the physical containment of the header components. That is, the first header component will render as a heading level 1. Any header component nested in the first header component will render as a heading level 2, and so on. You can manually override the heading level on individual header components using the `headerLevel` attribute.

For information about using the `panelHeader` component, see [How to Use the panelHeader Component](#).

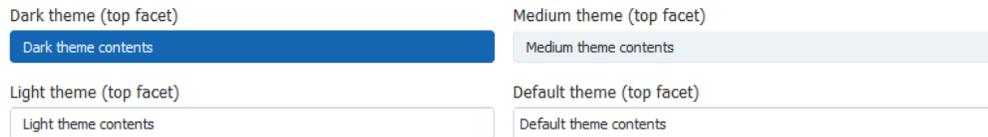
The `decorativeBox` component provides styling capabilities using themes. It has two facets, top and center. The top facet provides a noncolored area, while the center

facet is the actual box. The height of the top facet depends on whether or not a component has been put into the top facet. When the facet is set, the `topHeight` attribute is used to specify the size the content should occupy.

The color of the box for the center facet depends on the theme and skin used.

[Figure 9-68](#) shows the different themes available by default.

Figure 9-68 Themes Used in a decorativeBox Component



By default, the `decorativeBox` component stretches to fill its parent component. You can also configure the `decorativeBox` component to inherit its dimensions from its child components. For example, [Figure 9-69](#) shows the dark-theme `decorativeBox` configured to stretch to fill its parent, while the medium-theme `decorativeBox` is configured to only be as big as its child `outputText` component.

Figure 9-69 decorativeBox Can Stretch or Not



How to Use the panelHeader Component

You can use one `panelHeader` component to contain specific information, or you can use a series of nested `panelHeader` components to create a hierarchical organization of content. If you want to be able to hide and display the content, use the `showDetailHeader` component instead. See [How to Use the showDetailHeader Component](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Items in a Static Box](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use a `panelHeader` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Header** onto the page.
2. In the Properties window, expand the **Appearance** section.
3. Set **Text** to the label you want to display for this panel.
4. To add an icon before the label, set **Icon** to the URI of the image file to use.

 **Note:**

Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

5. If you are using the header to provide specific messaging information, set **MessageType** to one of the following values:
 - **confirmation**: The confirmation icon (represented by a note page overlaid with a green checkmark) replaces any specified icon image.
 - **error**: The error icon (represented by a red circle with an "x" inside) replaces any specified icon image. The header label also changes to red.
 - **info**: The info icon (represented by a blue circle with an "i" inside) replaces any specified icon image.
 - **none**: Default. No icon is displayed.
 - **warning**: The warning icon (represented by a yellow triangle with an exclamation mark inside) replaces any specified icon image.

[Figure 9-70](#) shows the icons used for the different message types.

Figure 9-70 Icons for Message Types

Error	
Warning	
Confirmation	
Info	

 **Note:**

Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

6. To display help for the header, enter the topic ID for **HelpTopicId**. For information about creating and using help topics, see [Displaying Help for Components](#).
7. To override the heading level for the component, set **headerLevel** to the desired level, for example H1, H2, etc. through H6.

The heading level is used to determine the correct page structure, especially when used with screen reader applications. By default, **headerLevel** is set to -1, which allows the headers to determine their size based on the physical location on the page. In other words, the first header component will be set to be a H1. Any header component nested in that H1 component will be set to H2, and so on.

 **Note:**

Screen reader applications rely on the HTML header level assignments to identify the underlying structure of the page. Make sure your use of header components and assignment of header levels make sense for your page.

When using an override value, consider the effects of having headers inside disclosable sections of the page. For example, if a page has collapsible areas, you need to be sure that the overridden structure will make sense when the areas are both collapsed and disclosed.

8. If you want to change just the size of the header text, and not the structure of the heading hierarchy, set the `size` attribute.

The `size` attribute specifies the number to use for the header text and overrides the skin. The largest number is 0, and it corresponds to an H1 header level; the smallest is 5, and it corresponds to an H6 header.

By default, the `size` attribute is -1. This means ADF Faces automatically calculates the header level style to use from the topmost, parent component. When you use nested components, you do not have to set the `size` attribute explicitly to get the proper header style to be displayed.

 **Note:**

While you can force the style of the text using the `size` attribute, (where 0 is the largest text), the value of the `size` attribute will not affect the hierarchy. It only affects the style of the text.

In the default skin used by ADF Faces, the style used for sizes above 2 will be displayed the same as size 2. You can change this by creating a custom skin. See [What You May Need to Know About Skinning and the panelHeader Component](#).

9. If you want to control how the `panelHeader` component handles geometry management, expand the Appearance section and set **Type** to one of the following. For information about geometry management, see [Geometry Management and Component Stretching](#).
 - **flow**: The component will not stretch or stretch its children. The height of the `panelHeader` component will be determined solely by its children.
 - **stretch**: The component will stretch and stretch its child (will only stretch a single child component).
 - **default**: if you want the parent component of the `panelHeader` component to determine geometry management.
10. To add toolbar buttons to a panel, insert the `toolbar` component into the `toolbar` facet. Then, insert the desired number of button components into the `toolbar` component. For information about using `toolbar` and buttons, see [Using Toolbars](#).

**Note:**

Toolbar overflow is not supported in `panelHeader` components.

11. To add menus to a panel, insert menu components into the `menuBar` facet. For information about creating menus in a menu bar, see [Using Menus in a Menu Bar](#).

**Tip:**

You can place menus in the `toolbar` facet and toolbars (and toolboxes) in the `menu` facet. The main difference between these facets is location. The `toolbar` facet is before the `menu` facet.

12. Add contents to the other facets as needed.

**Tip:**

If any facet is not visible in the visual editor:

- a. Right-click the `panelHeader` component in the Structure window.
- b. From the context menu, choose **Facets - Panel Header >facet name**. Facets in use on the page are indicated by a checkmark in front of the facet name.

13. To add contents to the panel, insert the desired child components into the `panelHeader` component.

How to Use the `decorativeBox` Component

You use the `decorativeBox` component to provide a colored area or box in a page. This component is typically used as a container for the `navigationPane` component that is configured to display tabs. See [Using Navigation Items for a Page Hierarchy](#).

To create and use a `decorativeBox` component:

1. In the Components window, from the Layout panel, drag and drop a **Decorative Box** onto the page.
2. In the Properties window, expand the **Common** section and set **Top Height** to the height for the `top` facet.
3. To change the theme, expand the **Style and Theme** section and choose a different theme.
4. By default, the `decorativeBox` component stretches to fill available browser space. If instead, you want to use the `decorativeBox` component as a child to a component that does not stretch its children, then you need to change how the `decorativeBox` component handles stretching.

You configure whether the component will stretch or not using the `dimensionsFrom` attribute.

Note:

The default value for the `dimensionsFrom` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want the components whose geometry management is determined by the `dimensionsFrom` attribute to stretch if its parent component allows stretching of its child, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `dimensionsFrom` attribute. Set the `dimensionsFrom` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `dimensionsFrom` is based on the component's default value, as documented in the following descriptions. See [Geometry Management for Layout and Table Components](#).

Set `DimensionsFrom` to one of the following:

- `children`: the `decorativeBox` component will get its dimensions from its child components.

 **Note:**

If you use this setting, you cannot use a percentage to set the height of the `top` facet. If you do, the `top` facet will try to get its dimensions from the size of this `decorativeBox` component, which will not be possible, as the `decorativeBox` component will be getting its height from its contents, resulting in a circular dependency. If a percentage is used, it will be disregarded and the default `50px` will be used instead.

Similarly, you cannot set the height of the `decorativeBox` (for example through the `inlineStyle` or `styleClass` attributes). Doing so would cause conflict between the `decorativeBox` height and the child component height.

- `parent`: the size of the `decorativeBox` component will be determined in the following order:
 - From the `inlineStyle` attribute.
 - If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.
- `auto`: If the parent component to the `decorativeBox` component allows stretching of its child, then the `decorativeBox` component will stretch to fill the parent. If the parent does not stretch its children then the size of the `decorativeBox` component will be based on the size of its child component.

See [What You May Need to Know About Geometry Management and the `decorativeBox` Component](#).

What You May Need to Know About Geometry Management and the `decorativeBox` Component

The `decorativeBox` component can stretch child components in its `center` facet and it can also be stretched. The following components can be stretched inside the `center` facet of the `decorativeBox` component:

- `inputText` (when configured to stretch)
- `deck`
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection` (when configured to stretch)
- `panelDashboard`
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelLabelAndMessage` (when configured to stretch)

- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tableLayout` (when configured to stretch. Note that this is a MyFaces Trinidad component)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a facet of the `decorativeBox` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelHeader`
- `panelList`
- `showDetail`
- `showDetailHeader`

You cannot place components that cannot stretch into facets of a component that stretches its child components. Therefore, if you need to place one of the components that cannot be stretched into a facet of the `decorativeBox` component, wrap that component in a transition component that does not stretch its child components.

For example, if you want to place content in a `panelBox` component and have it flow within a facet of the `decorativeBox` component, you could place a `panelGroupLayout` component with its `layout` attribute set to `scroll` in the facet of the `decorativeBox` component, and then place the `panelBox` component in that `panelGroupLayout` component. See [Nesting Components Inside Components That Allow Stretching](#).

What You May Need to Know About Skinning and the `panelHeader` Component

The `panelHeader` component uses styles specified in the application's skin to determine its heading sizes. Heading sizes above 2 will be displayed the same as size 2. That is, there is no difference in styles for sizes 3, 4, or 5—they all show the same style as size 2. You can change this by creating a custom skin.

See [Customizing the Appearance Using Styles and Skins](#).

What You May Need to Know About Skinning and the `decorativeBox` Component

A `decorativeBox` component that renders using the Skyros skin uses themes and skinning keys to control the borders and colors of its different facets. For example, if

you use the default theme, the `decorativeBox` component body is white and the border is blue, and the top-left corner is rounded. If you use the medium theme, the body is a medium blue.

 **Note:**

If you use the simple borders feature of the Skyros skin, then certain border elements, such as corners, are not rendered at all.

You can further control the style of the `decorativeBox` component using skins. Skinning keys can be defined for the following areas of the component:

- top-start
- top
- top-end
- start
- end
- bottom-start
- bottom
- bottom-end

The Alta skin does not use themes. See [Customizing the Appearance Using Styles and Skins](#).

Displaying a Bulleted List in One or More Columns

Using the ADF Faces `panelList` component, you can display the content as a bulleted list. You can nest the `panelList` component by wrapping them in a group component. The `panelList` component is a layout element for displaying a vertical list of child components with a bullet next to each child, as shown in [Figure 9-71](#). Only child components whose `rendered` attribute is set to `true` and whose `visible` attribute is set to `true` are considered for display by in the list.

 **Note:**

To display dynamic data (for example, a list of data determined at runtime by JSF bindings), use the selection components, as documented in [Using Selection Components](#). If you need to create lists that change the model layer, see [Using List-of-Values Components](#).

Figure 9-71 PanelList Component with Default Disc Bullet

- outputText1
- outputText2
- outputText3
- [commandLink 1](#)
- [commandLink 2](#)

By default, the disc bullet is used to style the child components. There are other styles you can use, such as square bullets and white circles. You can also split the list into columns when you have a very long list of items to display.

How to Use the panelList Component

Use one `panelList` component to create each list of items.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying a Bulleted List in One or More Columns](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelList` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel List** onto the JSF page.
2. In the Properties window, expand the **Common** section, and set the `listStyle` attribute to a valid CSS 2.1 list style value, such as one of the following:
 - `list-style-type: disc`
 - `list-style-type: square`
 - `list-style-type: circle`
 - `list-style-type: decimal`
 - `list-style-type: lower-alpha`
 - `list-style-type: upper-alpha`

For example, the `list-style-type: disc` attribute value corresponds to a disc bullet, and the `list-style-type: circle` value corresponds to a circle bullet.

For a complete list of the valid style values to use, refer to the CSS 2.1 Specification for generated lists at

<http://www.w3.org/TR/CSS21/generate.html>

 **Tip:**

Some browsers support more style options than others, for example, `upper-roman`, `lower-roman`, and `lower-greek`. Use of these is cautioned because they will not display consistently across web browsers.

The following example shows the code for setting the list style to a circle.

```
<af:panelList listStyle="list-style-type: circle" ...>  
  <!-- child components here -->  
</af:panelList>
```

3. Insert the desired number of child components (to display as bulleted items) into the `panelList` component.

 **Tip:**

Panel lists also allow you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the panel list.

For example, you could insert a series of `commandLink` components or `outputFormatted` components.

 **Note:**

By default, ADF Faces displays all rendered child components of a `panelList` component in a single column. For details on how to split the list into two or more columns and for information about using the `rows` and `maxColumns` attributes, see [Arranging Content in Forms](#). The concept of using the `rows` and `maxColumns` attributes for columnar display in the `panelList` and `panelFormLayout` components are the same.

What You May Need to Know About Creating a List Hierarchy

You can nest `panelList` components to create a list hierarchy. A list hierarchy, as shown in [Figure 9-72](#), has outer items and inner items, where the inner items belonging to an outer item are indented under the outer item. Each group of inner items is created by one nested `panelList` component.

Figure 9-72 Hierarchical List Created Using Nested panelList Components

- [item 1](#)
 - [item 1.1](#)
 - [item 1.2](#)
 - [item 1.3](#)
 - [item 1.4](#)
- [item 2](#)
 - [item 2.1](#)
 - [item 2.2](#)

To achieve the list hierarchy as shown in [Figure 9-72](#), use a `group` component to wrap the components that make up each group of outer items and their respective inner items. The following example shows the code for how to create a list hierarchy that has one outer item with four inner items, and another outer item with two inner items.

```
<af:panelList>
  <!-- First outer item and its four inner items -->
  <af:group>
    <af:commandLink text="item 1"/>
    <af:panelList>
      <af:commandLink text="item 1.1"/>
      <af:commandLink text="item 1.2"/>
      <af:commandLink text="item 1.3"/>
      <af:commandLink text="item 1.4"/>
    </af:panelList>
  </af:group>
  <!-- Second outer item and its two inner items -->
  <af:group>
    <af:commandLink text="item 2"/>
    <af:panelList>
      <af:commandLink text="item 2.1"/>
      <af:commandLink text="item 2.2"/>
    </af:panelList>
  </af:group>
</af:panelList>
```

By default, the outer list items (for example, item 1 and item 2) are displayed with the disc bullet, while the inner list items (for example, item 1.1 and item 2.1) have the white circle bullet.

For information about the `panelGroupLayout` component, see [Grouping Related Items](#).

Grouping Related Items

Using the ADF Faces `group` or `panelGroupLayout` component, you can form a group of similar or related components. The `group` component is just a container that can group related components together, whereas the `panelGroupLayout` provides a layout for its child components.

To keep like items together within a parent component, use either the `group` or `panelGroupLayout` component. The `group` component aggregates or groups together child components that are related semantically. Unlike the `panelGroupLayout` component, the `group` component does not provide any layout for its child

components. Used on its own, the `group` component does not render anything; only the child components inside of a `group` component render at runtime.

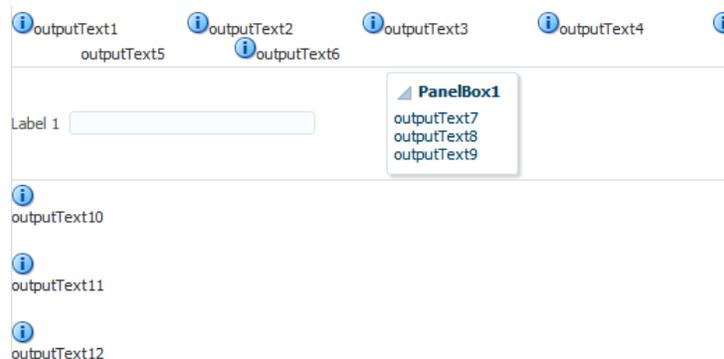
You can use any number of `group` components to group related components together. For example, you might want to group some of the input fields in a form layout created by the `panelFormLayout` component.

The following example shows sample code that groups two sets of child components inside a `panelFormLayout` component.

```
<af:panelFormLayout>
  <af:inputDate label="Pick a date"/>
  <!-- first group -->
  <af:group>
    <af:selectManyCheckbox label="Select all that apply">
      <af:selectItem label="Coffee" value="1"/>
      <af:selectItem label="Cream" value="1"/>
      <af:selectItem label="Low-fat Milk" value="1"/>
      <af:selectItem label="Sugar" value="1"/>
      <af:selectItem label="Sweetener"/>
    </af:selectManyCheckbox>
    <af:inputText label="Special instructions" rows="3"/>
  </af:group>
  <!-- Second group -->
  <af:group>
    <af:inputFile label="File to upload"/>
    <af:inputText label="Enter passcode"/>
  </af:group>
  <af:inputText label="Comments" rows="3"/>
  <af:spacer width="10" height="15"/>
  <f:facet name="footer"/>
</af:panelFormLayout>
```

The `panelGroupLayout` component lets you arrange a series of child components vertically or horizontally without wrapping, or consecutively with wrapping, as shown in [Figure 9-73](#). The `layout` attribute value determines the arrangement of the child components.

Figure 9-73 `panelGroupLayout` Arrangements



In all arrangements, each pair of adjacent child components can be separated by a line or white space using the `separator` facet of the `panelGroupLayout` component. See [Separating Content Using Blank Space or Lines](#).

When using the horizontal layout, the child components can also be vertically or horizontally aligned. For example, you could make a short component beside a tall component align at the top, as shown in [Figure 9-74](#).

Figure 9-74 Top-Aligned Horizontal Layout with `panelGroupLayout`

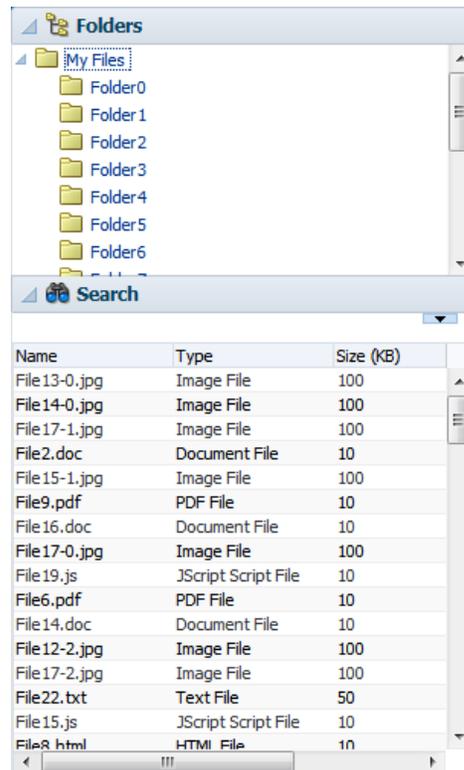


Unlike the `panelSplitter` or `panelStretchLayout` components, the `panelGroupLayout` component does not stretch its child components. Suppose you are already using a `panelSplitter` or `panelStretchLayout` component as the root component for the page, and you have a large number of child components to flow, but are not to be stretched. To provide scrollbars when flowing the child components, wrap the child components in the `panelGroupLayout` component with its `layout` attribute set to `scroll`, and then place the `panelGroupLayout` component inside a facet of the `panelSplitter` or `panelStretchLayout` component.

When the `layout` attribute is set to `scroll` on a `panelGroupLayout` component, ADF Faces automatically provides a scrollbar at runtime when the contents contained by the `panelGroupLayout` component are larger than the `panelGroupLayout` component itself. You do not have to write any code to enable the scrollbars, or set any inline styles to control the overflow.

For example, when you use layout components such as the `panelSplitter` component that let users display and hide child components contents, you do not have to write code to show the scrollbars when the contents are displayed, and to hide the scrollbars when the contents are hidden. Simply wrap the contents to be displayed inside a `panelGroupLayout` component, and set the `layout` attribute to `scroll`.

In the File Explorer application, the Search Navigator contains a `panelSplitter` component used to hide and show the search criteria. When the search criteria are hidden, and the search results content does not fit into the area, a scrollbar is rendered, as shown in [Figure 9-75](#).

Figure 9-75 Scrollbars Rendered Using `panelGroupLayout`**Note:**

If you include the `PanelGroupLayout` component while creating an emailable page, remove the `<td width="100%"></td>` entry within the `<table>... </table>` element in the html source. Otherwise, the footer row will not render properly in the email client. See [Creating Emailable Pages](#).

How to Use the `panelGroupLayout` Component

Any number of `panelGroupLayout` components can be nested to achieve the desired layout.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Grouping Related Items](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the `panelGroupLayout` component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Group Layout** onto the JSF page.
2. Insert the desired child components into the `panelGroupLayout` component.

 **Tip:**

The `panelGroupLayout` component also allows you to use the `iterator`, `switcher`, and `group` components as direct child components, providing these components wrap child components that would typically be direct child components of the `panelGroupLayout` component.

3. To add spacing or separator lines between adjacent child components, insert the `spacer` or `separator` component into the `separator` facet.
4. In the Properties window, expand the **Appearance** section. To arrange the child components in the desired layout, set **Layout** to one of the following values:
 - **default:** Provides consecutive layout with wrapping.

At runtime, when the contents exceed the browser space available (that is, when the child components are larger than the width of the parent container `panelGroupLayout`), the browser flows the contents onto the next line so that all child components are displayed.

 **Note:**

ADF Faces uses the bidirectional algorithm when making contents flow. Where there is a mix of right-to-left content and left-to-right content, this may result in contents not flowing consecutively.

- **horizontal:** Uses a horizontal layout, where child components are arranged in a horizontal line. No wrapping is provided when contents exceed the amount of browser space available.

In a horizontal layout, the child components can also be aligned vertically and horizontally. By default, horizontal child components are aligned in the center with reference to an imaginary horizontal line, and aligned in the middle with reference to an imaginary vertical line. To change the horizontal and vertical alignments of horizontal components, use the following attributes:

- **halign:** Sets the horizontal alignment. The default is `center`. Other acceptable values are: `start`, `end`, `left`, `right`.

For example, set `halign` to `start` if you want horizontal child components to always be left-aligned in browsers where the language reading direction is left-to-right, and right-aligned in a right-to-left reading direction.

- **valign:** Sets the vertical alignment. Default is `middle`. Other acceptable values are: `top`, `bottom`, `baseline`.

In output text components (such as `outputText`) that have varied font sizes in the text, setting `valign` to `baseline` would align the letters of the text along an imaginary line on which the letters sit, as shown in [Figure 9-76](#). If you set `valign` to `bottom` for such text components, the resulting effect would not be as pleasant looking, because `bottom` vertical alignment causes the bottommost points of all the letters to be on the same imaginary line.

Figure 9-76 Bottom and Baseline Vertical Alignment of Text

baseline valign outputText2

bottom valign outputText2

 **Note:**

The `halign` and `valign` attributes are ignored if the layout is not horizontal.

- **scroll:** Uses a vertical layout, where child components are stacked vertically, and a vertical scrollbar is provided when necessary.
- **vertical:** Uses a vertical layout, where child components are stacked vertically.

What You May Need to Know About Geometry Management and the `panelGroupLayout` Component

While the `panelGroupLayout` component cannot stretch its child components, it can be stretched when it is the child of a `panelSplitter` or `panelStretchLayout` component and its `layout` attribute is set to either `scroll` or `vertical`.

Separating Content Using Blank Space or Lines

Using the ADF Faces `spacer` component, you can include some blank space in a page. You can include vertical as well as horizontal space to even out the contents in a page. Use the `separator` component to include a horizontal line. You can incorporate some blank space in your pages, to space out the components so that the page appears less cluttered than it would if all the components were presented immediately next to each other, or immediately below each other. The ADF Faces component provided specifically for this purpose is the `spacer` component.

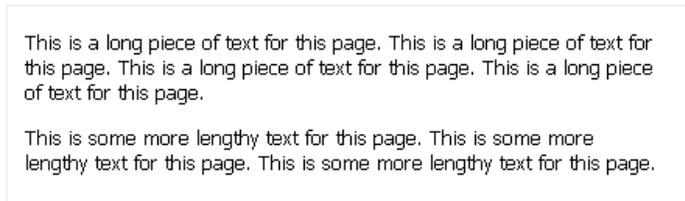
You can include either or both vertical and horizontal space in a page using the `height` and `width` attributes.

The `height` attribute determines the amount of vertical space to include in the page. The following example shows a page set up to space out two lengthy `outputText` components with some vertical space.

```
<af:panelGroupLayout layout="vertical">
  <af:outputText value="This is a long piece of text for this page..." />
  <af:spacer height="10" />
  <af:outputText value="This is some more lengthy text ..." />
</af:panelGroupLayout>
```

Figure 9-77 shows the effect the `spacer` component has on the page output as viewed in a browser.

Figure 9-77 Vertical Space Viewed in a Browser



The `width` attribute determines the amount of horizontal space to include between components. The following example shows part of the source of a page set up to space out two components horizontally.

```
<af:outputLabel value="Your credit rating is currently:"/>
<af:spacer width="10"/>
<af:outputText value="Level 8"/>
```

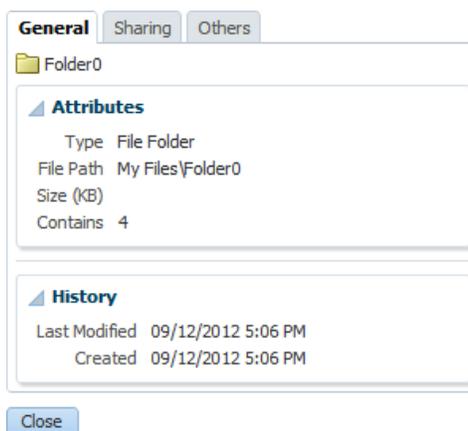
Figure 9-78 shows the effect of spacing components horizontally as viewed in a browser.

Figure 9-78 Horizontal Space Viewed in a Browser



The `separator` component creates a horizontal line. Figure 9-79 shows the `properties.jspx` file as it would be displayed with a `separator` component inserted between the two `panelBox` components.

Figure 9-79 Using the separator Component to Create a Line



The `spacer` and `separator` components are often used in facets of other layout components. Doing so ensures that the space or line stays with the components they were meant to separate.

How to Use the spacer Component

You can use as many `spacer` components as needed on a page.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Separating Content Using Blank Space or Lines](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the spacer component:

1. In the Components window, from the Layout panel, drag and drop a **Spacer** to the JSF page.
2. In the Properties window, expand the **Common** section. Set the width and height as needed.

Note:

If the height is specified but not the width, a block-level HTML element is rendered, thereby introducing a new line effect. If the width is specified, then, irrespective of the specified value of height, it may not get shorter than the applicable line-height in user agents that strictly support HTML standards.

How to Use the Separator Component

You can use as many `separator` components as needed on a page.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Separating Content Using Blank Space or Lines](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Layout Components](#).

To create and use the separator component:

1. In the Components window, from the Layout panel, drag and drop a **Separator** onto the JSF page.
2. In the Properties window, set the properties as needed.

10

Creating and Reusing Fragments, Page Templates, and Components

This chapter describes how to create reusable content and then use that content to build portions of JSF pages or entire pages. It describes how to use page templates to define entire page layouts that can be applied to pages. It also describes how to use page fragments to build complex pages. It then describes how to create reusable declarative components using existing ADF Faces components.

This chapter includes the following sections:

- [About Reusable Content](#)
- [Using Page Templates](#)
- [Using Page Fragments](#)
- [Using Declarative Components](#)
- [Adding Resources to Pages](#)

About Reusable Content

ADF Faces provides page templates, page fragments, and declarative components that you can use as reusable building blocks while designing a web page.

As you build JSF pages for your application, some pages may become complex and long, making editing complicated and tedious. Some pages may always contain a group of components arranged in a very specific layout, while other pages may always use a specific group of components in multiple parts of the page. And at times, you may want to share some parts of a page or entire pages with other developers. Whatever the case is, when something changes in the UI, you have to replicate your changes in many places and pages. Building and maintaining all those pages, and making sure that some sets or all are consistent in structure and layout can become increasingly inefficient.

Instead of using individual UI components to build pages, you can use page building blocks to build parts of a page or entire pages. The building blocks contain the frequently or commonly used UI components that create the reusable content for use in one or more pages of an application. Depending on your application, you can use just one type of building block, or all types in one or more pages. And you can share some building blocks across applications. When you modify the building blocks, the JSF pages that use the reusable content are automatically updated as well. Thus, by creating and using reusable content in your application, you can build web user interfaces that are always consistent in structure and layout, and an application that is scalable and extensible.

ADF Faces provides the following types of reusable building blocks:

- **Page templates:** By creating page templates, you can create entire page layouts using individual components and page fragments. For example, if you are repeatedly laying out some components in a specific way in multiple JSF pages, consider creating a page template for those pages. When you use the page

template to build your pages, you can be sure that the pages are always consistent in structure and layout across the application.

The page template and the declarative component share much of the functionality. The main difference is that the page template supports ADF Model binding and ADF Controller using a page template model. Using the `value` attribute, you can specify which object to use as the bindings inside of the page template. If the `value` is a page template model binding, ADF Model page bindings may be used, and you may use the ADF page definition to determine which view to include.

For information about creating and using page templates, see [Using Page Templates](#), and [How to Create JSF Pages Based on Page Templates](#).

- **Page fragments:** Page fragments allow you to create parts of a page. A JSF page can be made up of one or more page fragments. For example, a large JSF page can be broken up into several smaller page fragments for easier maintenance. For information about creating and using page fragments, see [Using Page Fragments](#).
- **Declarative components:** The declarative components feature allows you to assemble existing, individual UI components into one composite, reusable component, which you then declaratively use in one or more pages. For example, if you are always inserting a group of components in multiple places, consider creating a composite declarative component that comprises the individual components, and then reusing that declarative component in multiple places throughout the application. Declarative components can also be used in page templates.

The declarative component is deployed as part of an ADF library JAR file. It features its own TLD file that allows you to put the component in your own namespace. The declarative component allows you to pass facets into the component and also any attributes and method expressions. Inside of the declarative component, the attributes and facets may be accessed using EL expressions. It has a relatively low overhead as it does not involve ADF Model or ADF Controller, which also means that it does not have support for ADF Model transactions or ADF Controller page flows.

Note that you should not reference individual components inside of a declarative component, and individual components within a declarative component should not reference external components. The reason is that changes in the declarative component or in the consuming page could cause the partial triggers to no longer work. For information about creating and using declarative components, see [Using Declarative Components](#).

 **Tip:**

If your application uses ADF Controller and the ADF Model layer, then you can also use ADF regions. Regions used in conjunction with ADF bounded task flows, encapsulate business logic, process flow, and UI components all in one package, which can then be reused throughout the application. For complete information about creating and using ADF bounded task flows as regions, see *Using Task Flows as Regions in Developing Fusion Web Applications with Oracle Application Development Framework*.

Page templates, page fragments, and declarative components provide consistent structure and layout to the pages in an application. These building blocks can not only

be reused in the same application, but also can be shared across applications. When update a building block, all the instances where it is used is automatically updated.

Page templates are data-bound templates that support both static areas that do not change and dynamic areas where they change during runtime. You can use page fragments to build modular pages. For instance, you can create page fragments for the header, footer, and company logo and reuse these fragments throughout the application. You can use declarative components when you have several components that always used in a group. By creating a declarative component, you can add it to the tag library and be able to drag and drop the declarative component from the JDeveloper Components window.

Page templates, declarative components, and regions implement the `javax.faces.component.NamingContainer` interface. At runtime, in the pages that consume reusable content, the page templates, declarative components, or regions create component subtrees, which are then inserted into the consuming page's single, JSF component tree. Because the consuming page has its own naming container, when you add reusable content to a page, take extra care when using mechanisms such as `partialTargets` and `findComponent()`, as you will need to take into account the different naming containers for the different components that appear on the page. For information about naming containers, see [Locating a Client Component on a Page](#).

If you plan to include resources such as CSS or JavaScript, you can use the `af:resource` tag to add the resources to the page. If this tag is used in page templates and declarative components, the specified resources will be added to the consuming page during JSP execution. See [Adding Resources to Pages](#).

If you are not using an ADF task flow to navigate a portion of the page, you should not be using regions, but instead use one of the other compound components. Among the compound components, you should use a page template if you need to use bindings inside of your compound component and they differ from the bindings of the host page. You should use a declarative component if you do not need bindings for your page and do not need to use a bounded task flow as part of your page.

The view parts of a page (fragments, declarative components, and the main page) all share the same request scope. This may result in a collision when you use the same fragment or declarative component multiple times on a page, and when they share a backing bean. You should use `backingBeanScope` for declarative components and page templates. For information about scopes, see [Object Scope Lifecycles](#).

You can control whether child components of a page template or declarative component can be changed by external reference. For example, you can enable or disable the customization of the child components. Both `af:pageTemplateDef` and `af:componentDef` has a `definition` attribute that controls access. When `definition` is set to `public`, then the direct child components can be customized, while `definition` is set to `private`, the child components cannot be customized. The default value is `private`. You can modify `definition` by editing the source file or by using the Properties window.

See Customizing Applications with MDS in *Developing Fusion Web Applications with Oracle Application Development Framework* for information about customization.

Reusable Components Use Cases and Examples

The File Explorer application uses a `fileExplorerTemplate` to provide a consistent look and feel to all the pages in the application. The facets of the file provide working

area to place different types of information. The template defines an `appCopyright` facet that is used to display copyright information for every page.

The main page of the File Explorer application not only uses the page template, but also uses page fragments to contain the content for the individual facets of the template. The `header.jspx` page fragment contains the menu commands for the application.

If you have several components that works as a group and repeats in several places, you can define a declarative component to group these components together. Once you have created the component, you can use this declarative component like any other component. For example, you may use several `inputText` components to denote first name, last name, and email address. Since this three `inputText` components will be used repeatedly in your application, you can create a declarative component for them.

Additional Functionality for Reusable Components

You may find it helpful to understand other Oracle ADF features before you implement your reusable components. Following are links to other functionality that are related to reusable components.

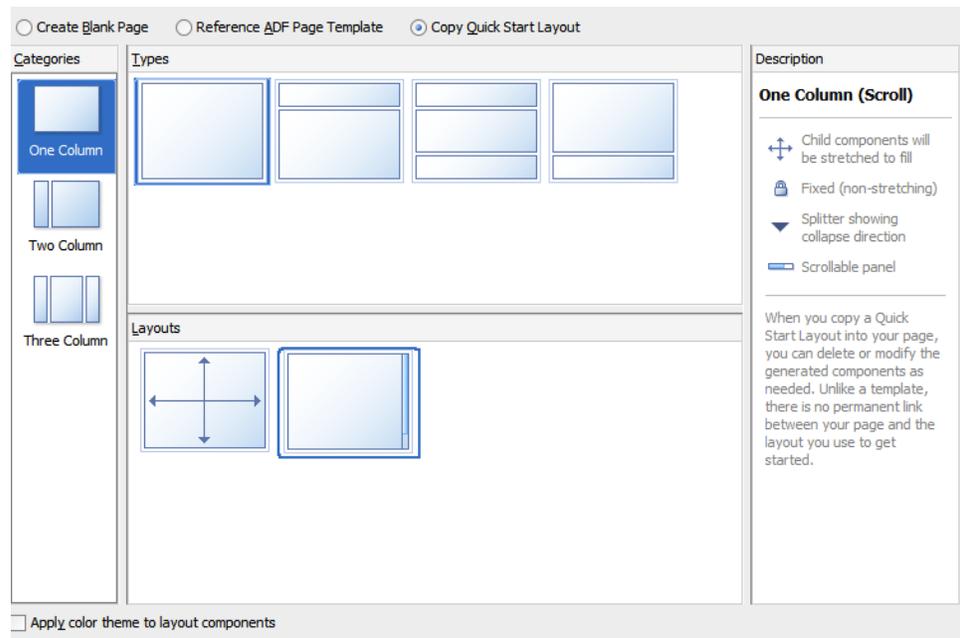
- For information about customization, see *Customizing Applications with MDS in Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about using the Quick Start Layouts to provide a preconfigured layout, see [Using Quick Start Layouts](#).
- For information about using model parameters and ADF Model data bindings, see *Using Page Templates in Developing Fusion Web Applications with Oracle Application Development Framework*.
- For information about packaging a page template into an ADF Library JAR file for reuse, see *Reusing Application Components in Developing Fusion Web Applications with Oracle Application Development Framework*.

Using Page Templates

Use an ADF Faces page template to define the layout for an entire page, which when used inherits the defined layout. You can specify fixed as well as dynamic content in a page template.

Page templates let you define entire page layouts, including values for certain attributes of the page. When pages are created using a template, they all inherit the defined layout. When you make layout modifications to the template, all pages that consume the template will automatically reflect the layout changes. You can either create the layout of your template yourself, or you can use one of the many quick layout designs. These predefined layouts automatically insert and configure the correct components required to implement the layout look and behavior you want. For example, you may want one column's width to be locked, while another column stretches to fill available browser space. [Figure 10-1](#) shows the quick layouts available for a two-column layout with the second column split between two panes. For information about the layout components, see [Organizing Content on Web Pages](#).

Figure 10-1 Quick Layouts



To use page templates in an application, you first create a page template definition. Page template definitions must be either Facelets or JSP XML documents because page templates embed XML content. In contrast to regular JSF pages where all components on the page must be enclosed within the `f:view` tag, page template definitions cannot contain an `f:view` tag and must have `pageTemplateDef` as the root tag. The page that uses the template must contain the `document` tag, (by default, JDeveloper adds the `document` tag to the consuming page).

A page template can have fixed content areas and dynamic content areas. For example, if a Help button should always be located at the top right-hand corner of pages, you could define such a button in the template layout, and when page authors use the template to build their pages, they do not have to add and configure a Help button. Dynamic content areas, on the other hand, are areas of the template where page authors can add contents within defined facets of the template or set property values that are specific to the type of pages they are building.

The entire description of a page template is defined within the `pageTemplateDef` tag, which has two sections. One section is within the `xmlContent` tag, which contains all the page template component metadata that describes the template's supported content areas (defined by facets), and available properties (defined as attributes). The second section (anything outside of the `xmlContent` tag) is where all the components that make up the actual page layout of the template are defined. The components in the layout section provide a JSF component subtree that is used to render the contents of the page template.

Facets act as placeholders for content on a page. In a page that consumes a template, page authors can insert content for the template only in named facets that have already been defined. This means that when you design a page template, you must define all possible facets within the `xmlContent` tag, using a `facet` element for each named facet. In the layout section of a page template definition, as you build the template layout using various components, you use the `facetRef` tag to reference the

named facets within those components where content can eventually be inserted into the template by page authors.

For example, the `fileExplorerTemplate` template contains a facet for copyright information and another facet for application information, as shown in the following example.

```
<facet> <description>
  <![CDATA[Area to put a link to more information
    about the application.]]>
</description>
<facet-name>appAbout</facet-name>
</facet>
<facet>
  <description>
    <![CDATA[The copyright region of the page. If present, this area
      typically contains an outputText component with the copyright
      information.]]>
  </description>
  <facet-name>appCopyright</facet-name>
</facet>
```

In the layout section of the template as shown in the following example, a `panelGroupLayout` component contains a table whose cell contains a reference to the `appCopyright` facet and another facet contains a reference to the `appAbout` facet. This is where a page developer will be allowed to place that content.

```
<af:panelGroupLayout layout="vertical">
  <afh:tableLayout width="100%">
    <afh:rowLayout>
      <afh:cellFormat>
        <af:facetRef facetName="appCopyright"/>
      </afh:cellFormat>
    </afh:rowLayout>
  </afh:tableLayout>
  <af:facetRef facetName="appAbout"/>
</af:panelGroupLayout>
```

Note:

Each named facet can be referenced only once in the layout section of the page template definition. That is, you cannot use multiple `facetRef` tags referencing the same `facetName` value in the same template definition.

While the `pageTemplateDef` tag describes all the information and components needed in a page template definition, the JSF pages that consume a page template use the `pageTemplate` tag to reference the page template definition. The following example shows how the `index.jspx` page references the `fileExplorerTemplate` template, provides values for the template's attributes, and places content within the template's facet definitions.

At design time, page developers using the template can insert content into the `appCopyright` facet, using the `f:facet` tag.

```
<af:pageTemplate id="fe"
  viewId="/fileExplorer/templates/fileExplorerTemplate.jspx">
  <f:attribute name="documentTitle"
```

```

        value="#{explorerBundle['global.branding_name'] }"/>
<f:attribute name="headerSize" value="70"/>
<f:attribute name="navigatorSize" value="370"/>
.
.
.
<f:facet name="appCopyright">
  <!-- Copyright info about File Explorer demo -->
  <af:outputFormatted value="#{explorerBundle['about.copyright'] }"/>
</f:facet>
.
.
.
</af:pageTemplate>

```

At runtime, the inserted content is displayed in the right location on the page, as indicated by `af:facetRef facetName="appCopyright"` in the template definition.

 **Note:**

You cannot run a page template as a run target in JDeveloper. You can run the page that uses the page template.

Page template attributes specify the component properties (for example, `headerGlobalSize`) that can be set or modified in the template. While `facet` element information is used to specify where in a template content can be inserted, `attribute` element information is used to specify what page attributes are available for passing into a template, and where in the template those attributes can be used to set or modify template properties. Page templates also support dynamic attributes as an inline tag. For example, `af:pageTemplate headerSize="70"` is valid syntax.

For the page template to reference its own attributes, the `pageTemplateDef` tag must have a `var` attribute, which contains an EL variable name for referencing each attribute defined in the template. For example, in the `fileExplorerTemplate` template, the value of `var` on the `pageTemplateDef` tag is set to `attrs`. Then in the layout section of the template, an EL expression such as `#{attrs.someAttributeName}` is used in those component attributes where page authors are allowed to specify their own values or modify default values.

For example, the `fileExplorerTemplate` template definition defines an attribute for the header size, which has a default int value of 100 pixels as shown in the following example.

```

<attribute>
  <description>
    Specifies the number of pixels tall that the global header content should
    consume.
  </description>
  <attribute-name>headerGlobalSize</attribute-name>
  <attribute-class>int</attribute-class>
  <default-value>100</default-value>
</attribute>

```

In the layout section of the template, the `splitterPosition` attribute of the `panelSplitter` component references the `headerGlobalSize` attribute in the EL expression `{attrs.headerGlobalSize}`, as shown in the following code:

```
<af:panelSplitter splitterPosition="{attrs.headerGlobalSize}" ../>
```

When page authors use the template, they can modify the `headerGlobalSize` value using `f:attribute`, as shown in the following code:

```
<af:pageTemplate ..>
  <f:attribute name="headerGlobalSize" value="50"/>
  .
  .
  .
</af:pageTemplate>
```

At runtime, the specified attribute value is substituted into the appropriate part of the template, as indicated by the EL expression that bears the attribute name.

 **Tip:**

If you define a resource bundle in a page template, the pages that consume the template will also be able to use the resource bundle. For information about using resource bundles, see [Manually Defining Resource Bundles and Locales](#).

You can nest templates when you need to reuse the same content across multiple templates. For example, say your application will have three different types of pages, but the header and footer will always be the same. Instead of having to include the same header and footer design in three different templates, you can create a header template and a footer template, and then simply nest those templates into each of the different page templates.

For a simple page template, it is probably sufficient to place all the components for the entire layout section into the page template definition file. For a more complex page template, you can certainly break the layout section into several smaller fragment files for easier maintenance, and use `jsp:include` tags to include and connect the various fragment files.

When you break the layout section of a page template into several smaller fragment files, all the page template component metadata must be contained within the `xmlContent` tag in the main page template definition file. There can be only one `xmlContent` tag within a `pageTemplateDef` tag. You cannot have page template component metadata in the fragment files; fragment files can contain portions of the page template layout components only.

 **Note:**

You cannot nest page templates inside other page templates.

If your template requires resources such as custom styles defined in CSS or JavaScript, then you need to include these on the consuming page, using the `af:resource` tag. See [Adding Resources to Pages](#).

How to Create a Page Template

JDeveloper simplifies creating page template definitions by providing the Create JSF Page Template wizard, which lets you add named facets and attributes declaratively to create the template component metadata section of a template. In addition to generating the metadata code for you, JDeveloper also creates and modifies a `pagetemplate-metadata.xml` file that keeps track of all the page templates you create in a project.

Performance Tip:

Because page templates may be present in every application page, templates should be optimized so that common overhead is avoided. One example of overhead is round corners, for example on boxes, which are quite expensive. Adding them to the template will add overhead to every page.

Before you begin:

It may be helpful to have an understanding of page templates. See [Using Page Templates](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To create a page template definition:

1. In the Applications window, right-click the node where you wish to create and store page templates and choose **New > ADF Page Template**.
2. In the Create a Page Template dialog, enter a file name for the page template definition.

Page template definitions must be XML documents (with file extension `.jspx`) because they embed XML content.

Performance Tip:

Avoid long names because they can have an impact on server-side, network traffic, and client processing.

3. Accept the directory name for the template definition, or choose a new location.
If the page template is intended to be packaged as an ADF Library, you should not accept the default directory name. You should try to specify a unique directory name so that it will be less likely to clash with page templates from other ADF Libraries.
4. Select either **Facelets** or **JSP XML** as the document type.

5. Enter a Page Template name for the page template definition, and click **Next**.
6. In the Optionally add starting content page of the Create a Page Template dialog, select one of the template choices, then the template, and click **Next**.
 - **Blank Template**: Select if you want start using a blank page.
 - **Copy Existing Template**: Select if you want to start with an existing template.
 - **Copy Quick Start Layout**: Select if you want to use various predefined templates featuring one, two, or three column layout.
7. In the Add facet definitions and attribute page of the Create a Page Template dialog, you can add facets and attributes.
 - In the Facet Definitions section, click **Add** to add a facet name and description.

Facets are predefined areas on a page template where content can eventually be inserted when building pages using the template. Each facet must have a unique name. For example, you could define a facet called `main` for the main content area of the page, and a facet called `branding` for the branding area of the page.

 **Tip:**

If you plan on nesting templates or using more than one template on a page, to avoid confusion, use unique names for the facets in all templates.

- In the Attributes section, click **Add** to add an attribute.

Attributes are UI component attributes that can be passed into a page when building pages using the template. Each attribute must have a name and class type (for example, `java.lang.String`). Note that whatever consumes the attribute (for example an attribute on a component that you configure in Step 13) must be able to accept that type. You can assign default values, and you can specify that the values are mandatory by selecting the `Required` checkbox.
 - Click **Next**.
8. If the page template contents use ADF Model data bindings, select the **Create Page Definition** checkbox, and click **Add** to add one or more model parameters. For information about using model parameters and ADF Model data bindings, see Using Page Templates in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Once you complete the wizard, JDeveloper displays the page template definition file in the visual editor. The following example shows the code JDeveloper adds for you when you use the wizard to define the metadata for a page template definition. You can view this code in the source editor.

**Tip:**

Once a template is created, you can add facets and attributes by selecting the `pageTemplateDef` tag in the Structure window and using the Properties window.

**Note:**

When you change or delete any facet name or attribute name in the template component metadata, you have to manually change or delete the facet or attribute name referenced in the layout section of the template definition, as well as the JSF pages that consume the template.

```
<af:pageTemplateDef var="attrs">
  <af:xmlContent>
    <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
      <description/>
      <display-name>sampleTemplateDef1</display-name>
      <facet>
        <facet-name>main</facet-name>
      </facet>
      .
      .
      .
      <attribute>
        <attribute-name>Title</attribute-name>
        <attribute-class>java.lang.String</attribute-class>
        <default-value>Replace title here</default-value>
        <required>true</required>
      </attribute>
      .
      .
      .
    </component>
  </af:xmlContent>
  .
  .
  .
</af:pageTemplateDef>
```

9. In the Components window, drag and drop a component to the page.

In the layout section of a page template definition (or in fragment files that contain a portion of the layout section), you cannot use the `f:view` tag, because it is already used in the JSF pages that consume page templates.

 **Best Practice Tip:**

You should not use the `document` or `form` tags in the template. While theoretically, template definitions can use the `document` and `form` tags, doing so means the consuming page cannot. Because page templates can be used for page fragments, which in turn will be used by another page, it is likely that the consuming page will contain these tags. You should never add a `document` tag to a page template.

You can add any number of components to the layout section. However, you should only have one root component in a template. If you did not choose to use one of the quick start layouts, then typically, you would add a panel component such as `panelStretchLayout` or `panelGroupLayout`, and then add the components that define the layout into the panel component. See [Organizing Content on Web Pages](#).

Declarative components and databound components may be used in the layout section. For information about using declarative components, see [Using Declarative Components](#). For information about using databound components in page templates, see *Using Page Templates in Developing Fusion Web Applications with Oracle Application Development Framework*.

10. To nest another template into this template, in the Components window, from the Layout panel, in the Core Structure group, drag and drop a **Template** onto the page.

 **Note:**

You cannot nest an ADF databound template in a template that does not use ADF data binding, or in a declarative component.

Additionally, a nested template cannot be used more than one per rendering. For example, it cannot be used as a child to a component that stamps its children, such as a table or tree.

11. In the Insert Template dialog, select the template that you want to nest.

 **Tip:**

The dialog displays all the templates that are included in the current project or that are provided in an ADF Library. For information about ADF Libraries, see *Reusing Application Components in Developing Fusion Web Applications with Oracle Application Development Framework*.

12. In the Components window, from the Layout panel, in the Core Structure group, drag a **Facet** and drop it to the page.

For example, if you have defined a `main` facet for the main content area on a page template, you might add the `facetRef` tag as a child in the `center` facet of `panelStretchLayout` component to reference the `main` facet. At design time, when

the page author drops content into the `main` facet, the content is placed in the correct location on the page as defined in the template.

When you use the `facetRef` tag to reference the appropriate named facet, JDeveloper displays the Insert Facet dialog. In that dialog, select a facet name from the dropdown list, or enter a facet name. If you enter a facet name that is not already defined in the component metadata of the page template definition file, JDeveloper automatically adds an entry for the new facet definition in the component metadata within the `xmlContent` tag.

 **Note:**

Each facet can be referenced only once in the layout section of the page template definition. That is, you cannot use multiple `facetRef` tags referencing the same `facetName` value in the same template definition.

 **Note:**

If you have nested another template into this template, you must create facet references for each facet in the nested template as well as this template.

13. To specify where attributes should be used in the page template, use the page template's `var` attribute value to reference the relevant attributes on the appropriate components in the layout section.

The `var` attribute of the `pageTemplateDef` tag specifies the EL variable name that is used to access the page template's own attributes. As shown in the following example, the default value of `var` used by JDeveloper is `attrs`.

For example, if you have defined a `title` attribute and added the `panelHeader` component, you might use the EL expression `#{attrs.title}` in the `text` value of the `panelHeader` component, as shown in the following code, to reference the value of `title`:

```
<af:panelHeader text="#{attrs.title}">
```

14. To include another file in the template layout, use the `jsp:include` tag wrapped inside the `subview` tag to reference a fragment file, as shown in the following code:

```
<f:subview id="secDecor">  
  <jsp:include page="fileExplorerSecondaryDecoration.jspx"/>  
</f:subview>
```

The included fragment file must also be an XML document, containing only `jsp:root` at the top of the hierarchy. For information about using fragments, see [How to Use a Page Fragment in a JSF Page](#).

By creating a few fragment files for the components that define the template layout, and then including the fragment files in the page template definition, you can split up an otherwise large template file into smaller files for easier maintenance.

What Happens When You Create a Page Template

Note:

If components in your page template use ADF Model data binding, or if you chose to associate an ADF page definition when you created the template, JDeveloper automatically creates files and folders related to ADF Model. For information about the files used with page templates and ADF Model data binding, see Using Page Templates in *Developing Fusion Web Applications with Oracle Application Development Framework*.

The first time you use the wizard to create a page template in a project, JDeveloper automatically creates the `pagetemplate-metadata.xml` file, which is placed in the `/ViewController/src/META-INF` directory in the file system.

For each page template that you define using the wizard, JDeveloper creates a page template definition file (for example, `sampleTemplateDef1.jspx`), and adds an entry to the `pagetemplate-metadata.xml` file. The following example shows an example of the `pagetemplate-metadata.xml` file.

```
<pageTemplateDefs xmlns="http://xmlns.oracle.com/adf/faces/rich/pagetemplate">
  <pagetemplate-jsp-ui-def>/sampleTemplateDef1.jspx</pagetemplate-jsp-ui-def>
  <pagetemplate-jsp-ui-def>/sampleTemplateDef2.jspx</pagetemplate-jsp-ui-def>
</pageTemplateDefs>
```

Note:

When you rename or delete a page template in the Applications window, JDeveloper renames or deletes the page template definition file in the file system, but you must manually change or delete the page template entry in the `pagetemplate-metadata.xml` file, and update or remove any JSF pages that use the template.

The `pagetemplate-metadata.xml` file contains the names and paths of all the page templates that you create in a project. This file is used to determine which page templates are available when you use a wizard to create template-based JSF pages, and when you deploy a project containing page template definitions.

How to Create JSF Pages Based on Page Templates

Typically, you create JSF pages in the same project where page template definitions are created and stored. If the page templates are not in the same project as where you are going to create template-based pages, first deploy the page templates project to an ADF Library JAR file. For information about deploying a project, see Reusing Application Components in *Developing Fusion Web Applications with Oracle Application Development Framework*. Deploying a page template project also allows you to share page templates with other developers working on the application.

 **Note:**

If the template uses `jsp:include` tags, then it cannot be deployed to an ADF Library to be reused in other applications.

You can use page templates to build JSF pages or page fragments. If you modify the layout section of a page template later, all pages or page fragments that use the template are automatically updated with the layout changes.

In the page that consumes a template, you can add content before and after the `pageTemplate` tag. In general, you would use only one `pageTemplate` tag in a page, but there are no restrictions for using more than one.

JDeveloper simplifies the creation of JSF pages based on page templates by providing a template selection option in the Create JSF Page or Create JSF Page Fragment wizard.

Before you begin:

It may be helpful to have an understanding of page templates. See [Using Page Templates](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To create a JSF page or page fragment based on a page template:

1. Follow the instructions in [How to Create JSF Pages](#) to open the Create JSF Page dialog. In the dialog, select a page template to use from the available selections.

 **Tip:**

Only page templates that have been created using the template wizard in JDeveloper are available for selection when you have selected **ReferenceADF Page Template**.

 **Tip:**

Instead of basing the whole page on a template, you can use a template for an area of a page. For example, you may have a template to be used just in the headers of your pages. To apply a template to an area of your page, from the Layout panel of the Components window, drag and drop a **Template** into the desired component.

By default, JDeveloper displays the new page or page fragment in the visual editor. The facets defined in the page template appear as named boxes in the visual editor. If the page template contains any default values, you should see the values in the Properties window, and if the default values have some visual representation (for example, size), that will be reflected in the visual editor, along

with any content that is rendered by components defined in the layout section of the page template definition.

2. In the Structure window, expand **jsp:root** until you see **af:pageTemplate** (which should be under **af:form**).

Within the `form` tag, you can drop content before and after the `pageTemplate` tag.

3. Add components by dragging and dropping components from the Components window in the facets of the template. In the Structure window, within **af:pageTemplate**, the facets (for example, **f:facet - main**) that have been predefined in the component metadata section of the page template definition are shown.

The type of components you can drop into a facet may be dependent on the location of the `facetRef` tag in the page template definition. For example, if you've defined a `facetRef` tag to be inside a `table` component in the page template definition, then only `column` components can be dropped into the facet because the `table` component accepts only `column` components as children.

 **Tip:**

The content you drop into the template facets may contain ADF Model data binding. In other words, you can drag and drop items from the Data Controls panel. For information about using ADF Model data binding, see *About Using ADF Model in a Fusion Web Application in Developing Fusion Web Applications with Oracle Application Development Framework*.

4. In the Structure window, select **af:pageTemplate**. Then, in the Properties window, you can see all the attributes that are predefined in the page template definition. Predefined attributes might have default values.

You can assign static values to the predefined attributes, or you can use EL expressions (for example, `#{myBean.somevalue}`). When you enter a value for an attribute, JDeveloper adds the `f:attribute` tag to the code, and replaces the attribute's default value (if any) with the value you assign.

At runtime, the default or assigned attribute value is used or displayed in the appropriate part of the template, as specified in the page template definition by the EL expression that bears the name of the attribute (such as `#{attrs.someAttributeName}`).

 **Note:**

In addition to predefined template definition attributes, the Properties window also shows other attributes of the `pageTemplate` tag such as `Id`, `Value`, and `ViewId`.

The `ViewId` attribute of the `pageTemplate` tag specifies the page template definition file to use in the consuming page at runtime. JDeveloper automatically assigns the `ViewId` attribute with the appropriate value when you use the wizard to create a template-based JSF page. The `ViewId` attribute value cannot be removed, otherwise a runtime error will occur, and the parts of the page that are based on the template will not render.

- To include resources, such as CSS or JavaScript, you need to use the `af:resource` tag. See [Adding Resources to Pages](#).

What Happens When You Use a Template to Create a Page

When you create a page using a template, JDeveloper inserts the `pageTemplate` tag, which references the page template definition, as shown in the following example. Any components added inside the template's facets use the `f:facet` tag to reference the facet. Any attribute values you specified are shown in the `f:attribute` tag.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <f:view>
    <af:document>
      <af:form>
        .
        .
        .
        <af:pageTemplate viewId="/sampleTemplateDef1.jspx" id="template1">
          <f:attribute name="title" value="Some Value"/>
          <f:facet name="main">
            <!-- add contents here -->
          </f:facet>
        </af:pageTemplate>
        .
        .
        .
      </af:form>
    </af:document>
  </f:view>
</jsp:root>
```

What Happens at Runtime: How Page Templates Are Resolved

When a JSF page that consumes a page template is executed:

- The `pageTemplate` component in the consuming page, using the `viewId` attribute (for example, `<af:pageTemplate viewId="/sampleTemplateDef1.jspx"/>`),

locates the page template definition file that contains the template component metadata and layout.

- The component subtree defined in the layout section of the `pageTemplateDef` tag is instantiated and inserted into the consuming page's component tree at the location identified by the `pageTemplate` tag in the page.
- The consuming page passes facet contents into the template using the `facet` tag. The facet contents of each `facet` tag are inserted into the appropriate location on the template as specified by the corresponding `facetRef` tag in the layout section of the `pageTemplateDef` tag.
- The consuming page passes values into the template by using the `attribute` tag. The `pageTemplateDef` tag sets the value of the `var` attribute so that the `pageTemplate` tag can internally reference its own parameters. The `pageTemplate` tag just sets the parameters; the runtime maps those parameters into the attributes defined in the `pageTemplateDef` tag.
- Using template component metadata, the `pageTemplate` tag applies any default values to its attributes and checks for required values.

 **Note:**

Page templates are processed during JSP execution, not during JSF processing (that is, component tree creation). This means that fragments built from page templates cannot be used within tags that require the component tree creation. For example, you could not include a fragment based on a template within an `iterator` tag and expect it to be included in a loop.

For information about what happens when the page template uses ADF Model data binding, see *Using Page Templates in Developing Fusion Web Applications with Oracle Application Development Framework*.

What You May Need to Know About Page Templates and Naming Containers

The `pageTemplate` component acts as a naming container for all content in the template (whether it is direct content in the template definition, or fragment content included using the `jsp:include` action). When working in template-based pages, you should not reference an individual component inside a page template. Changes made to the page template or its consuming page may cause the partial triggers to work improperly. See [What You May Need to Know About Using Naming Containers](#).

Using Page Fragments

You can break up complex pages into page fragments, which are incomplete JSF pages and reuse them in multiple pages. Different portions of a page can be created as page fragments and a page can use multiple page fragments. As you build web pages for an application, some pages may quickly become large and unmanageable. One possible way to simplify the process of building and maintaining complex pages is to use page fragments.

Large, complex pages broken down into several smaller page fragments are easier to maintain. Depending on how you design a page, the page fragments created for one page may be reused in other pages. For example, suppose different parts of several pages use the same form, then you might find it beneficial to create page fragments containing those components in the form, and reuse those page fragments in several pages. Deciding on how many page fragments to create for one or more complex pages depends on your application, the degree to which you wish to reuse portions of a page between multiple pages, and the desire to simplify complex pages.

Page fragments are incomplete JSF pages. A complete JSF page that uses ADF Faces must have the `document` tag enclosed within an `f:view` tag. The contents for the entire page are enclosed within the `document` tag. A page fragment, on the other hand, represents a portion of a complete page, and does not contain the `f:view` or `document` tags. The contents for the page fragment are simply enclosed within a `jsp:root` tag.

When you build a JSF page using page fragments, the page can use one or more page fragments that define different portions of the page. The same page fragment can be used more than once in a page, and in multiple pages.

 **Note:**

The view parts of a page (fragments, declarative components, and the main page) all share the same request scope. This may result in a collision when you use the same fragment or declarative component multiple times on a page and the fragments or components share a backing bean. For information about scopes, see [Object Scope Lifecycles](#).

For example, the File Explorer application uses one main page (`index.jspx`) that includes the following page fragments:

- `popups.jspx`: Contains all the popup code used in the application.
- `help.jspx`: Contains the help content.
- `header.jspx`: Contains the toolbars and menus for the application.
- `navigators.jspx`: Contains the tree that displays the node hierarchy of the application.
- `contentViews.jspx`: Contains the content for the node selected in the navigator pane.

The following example shows the abbreviated code for the included `header.jspx` page fragment. Note that it does not contain an `f:view` or `document` tag.

```
<?xml version='1.0' encoding='UTF-8'?>
<ui:composition xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
    xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
    xmlns:f="http://java.sun.com/jsf/core">
  <af:panelStretchLayout id="headerStretch">
    <f:facet name="center">
      <!-- By default, every toolbar is placed on a new row -->
      <af:toolbox id="headerToolbox"
        binding="#{explorer.headerManager.headerToolbox}">
        .
        .
      </af:toolbox>
    </f:facet>
  </af:panelStretchLayout>
</ui:composition>
```

```

    </af:toolbox>
  </f:facet>
</af:panelStretchLayout>
</ui:composition>

```

When you consume a page fragment in a JSF page, at the part of the page that will use the page fragment contents, you insert the `jsp:include` tag to include the desired page fragment file, as shown in the following example, which is abbreviated code from the `index.jspx` page.

```

<?xml version='1.0' encoding='utf-8'?>
<ui:composition xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
  xmlns:trh="http://myfaces.apache.org/trinidad/html">
  <jsp:directive.page contentType="text/html;charset=utf-8"/>
  <f:view>
  .
  .
  .
  <af:document id="fileExplorerDocument"
    title="#{explorerBundle['global.branding_name']}">
    <af:form id="mainForm">
      <!-- Popup menu definition -->
      <jsp:include page="/fileExplorer/popups.jspx"/>
      <jsp:include page="/fileExplorer/help.jspx"/>
      .
      .
      .
      <f:facet name="header">
        <af:group>
        <!-- The file explorer header with all the menus and toolbar buttons -->
        <jsp:include page="/fileExplorer/header.jspx"/>
        </af:group>
      </f:facet>
      <f:facet name="navigators">
        <af:group>
        <!-- The auxiliary area for navigating the file explorer -->
        <jsp:include page="/fileExplorer/navigators.jspx"/>
        </af:group>
      </f:facet>
      <f:facet name="contentViews">
        <af:group>
        <!-- Show the contents of the selected folder in the folders navigator -->
        <jsp:include page="/fileExplorer/contentViews.jspx"/>
        </af:group>
      </f:facet>
      .
      .
      .
    </af:form>
  </af:document>
  </f:view>
</ui:composition>

```

When you modify a page fragment, the pages that consume the page fragment are automatically updated with the modifications. With pages built from page fragments, when you make layout changes, it is highly probable that modifying the page

fragments alone is not sufficient; you may also have to modify every page that consumes the page fragments.

 **Note:**

If the consuming page uses ADF Model data binding, the included page fragment will use the binding container of the consuming page. Only page fragments created as part of ADF bounded task flows can have their own binding container. For information about ADF bounded task flows, see *Getting Started with ADF Task Flows in Developing Fusion Web Applications with Oracle Application Development Framework*.

Like complete JSF pages, page fragments can also be based on a page template, as shown in the following example. For information about creating and applying page templates, see [Using Page Templates](#), and [How to Create JSF Pages Based on Page Templates](#).

```
<?xml version='1.0' encoding='UTF-8'?>
<ui:composition xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
    xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
    xmlns:f="http://java.sun.com/jsf/core">
  <af:pageTemplate viewId="/someTemplateDefinition.jspx">
    .
    .
    .
  </af:pageTemplate>
</ui:composition>
```

How to Create a Page Fragment

Page fragments are just like any JSF page, except you do not use the `f:view` or `document` tags in page fragments. You can use the Create JSF Page Fragment wizard to create page fragments. When you create page fragments using the wizard, JDeveloper uses the extension `.jsff` for the page fragment files. If you do not use the wizard, you can use `.jspx` as the file extension (as the File Explorer application does); there is no special reason to use `.jsff` other than quick differentiation between complete JSF pages and page fragments when you are working in the Applications window in JDeveloper.

Before you begin:

It may be helpful to have an understanding of page fragments. See [Using Page Fragments](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To create a page fragment:

1. In the Applications window, right-click the node where you wish to create and store page fragments and choose **New > ADF Page Fragment**.
2. In the Create ADF Page Fragment dialog, enter a name for the page fragment file.
3. Accept the default directory for the page fragment, or choose a new location.

By default, JDeveloper saves page fragments in the project's `/public_html` directory in the file system. For example, you could change the default directory to `/public_html/fragments`.

4. You can have your fragment pre-designed for you by using either a ADF page template, a Quick Start Layout. or start with a blank page.
 - If you want to create a page fragment based on a page template, select the **Reference ADF Page Template** radio button and then select a template name from the dropdown list. For information about using page templates, see [How to Create JSF Pages Based on Page Templates](#).
 - If you want to use a Quick Start Layout, select the **Copy Quick Start Layout** radio button and browse to select the layout you want your fragment to use. Quick Start Layouts provide the correctly configured layout components need to achieve specific behavior and look. See [Using Quick Start Layouts](#).

When the page fragment creation is complete, JDeveloper displays the page fragment file in the visual editor.

5. To define the page fragment contents, drag and drop the desired components from the Components window onto the page.

You can use any ADF Faces or standard JSF component, for example `table`, `panelHeader`, or `f:facet`.

The following example shows an example of a page fragment that contains a menu component.

```
<?xml version='1.0' encoding='UTF-8'?>
<ui:composition xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
    xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
<af:pageTemplate viewId="/MytemplateDef1.jspx" id="pt1">
  <!-- page fragment contents start here -->
  <af:menu id="viewMenu"
    <af:group>
      <af:commandMenuItem type="check" text="Folders"/>
      <af:commandMenuItem type="check" text="Search"/>
    </af:group>
    <af:group>
      <af:commandMenuItem type="radio" text="Table"/>
      <af:commandMenuItem type="radio" text="Tree Table"/>
      <af:commandMenuItem type="radio" text="List"/>
    </af:group>
    <af:commandMenuItem text="Refresh"/>
  </menu>
</ui:composition>
```

What Happens When You Create a Page Fragment

In JDeveloper, because page fragment files use a different file extension from regular JSF pages, configuration entries are added to the `web.xml` file for recognizing and interpreting `.jsff` files in the application. The following example shows the `web.xml` configuration entries needed for `.jsff` files, which JDeveloper adds for you when you first create a page fragment using the wizard.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsff</url-pattern>
    <is-xml>>true</is-xml>
```

```
</jsp-property-group>  
</jsp-config>
```

By specifying the `url-pattern` subelement to `*.jsff` and setting the `is-xml` subelement to `true` in a `jsp-property-group` element, the application will recognize that files with extension `.jsff` are actually JSP documents, and thus must be interpreted as XML documents.

How to Use a Page Fragment in a JSF Page

To consume a page fragment in a JSF page, add the page using the Components window. You can use the `jsp:include` tag to include the desired page fragment file.

Before you begin:

It may be helpful to have an understanding of page fragments. See [Using Page Fragments](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To add a page fragment using the Components Window:

1. In the Components window, in the JSP page, drag a **Include** and drop it on the page.
2. In the Insert Include dialog, use the dropdown list to select the JSF page to include. Optionally, select whether or not to flush the buffer before the page is included. For help with the dialog, click **Help** or press F1.

What Happens at Runtime: How Page Fragments are Resolved

When the page that contains the included page(s) is executed, the `jsp:include` tag evaluates the view ID during JSF tree component build time and dynamically adds the content to the parent page at the location of the `jsp:include` tag. The fragment becomes part of the parent page after the component tree is built.

What You May Need to Know About Accessing a Page From a Different View ID

In ADF, the pages are accessed through different views, each with different viewID. Saved Search, layouts feature uses viewID to derive a unique location within MDS to save the layout files as binaries. If you open a page template or fragment that contains a component and attributes and its results from different viewIDs, the results layout will be saved in different locations in MDS, because the storage location depends on the viewID of the page. However, this breaks the results layout persistence feature in ADF Faces. To override the MDS location for one viewID with another viewID of the page containing results layout and to consolidate all layout files under one viewID, the public interface `QueryResultsLayoutIdentifier` should be registered with `ConfigPropertyService` key `oracle.adf.view.faces.component.query.resultsLayoutIdentifier` as given below.

```
public interface QueryResultsLayoutIdentifier  
{
```

```
    public abstract String getViewId(String currentViewId);  
}
```

Also add the ADF Share configuration `Configuration Property Service` key in `web.xml` or as an ADF share property and set the value as `oracle.adf.view.faces.component.query.resultsLayoutIdentifier`.

 **Note:**

You need to implement the public interface to return the correct viewId to be used.

Using Declarative Components

You can create an XML-based declarative component definition that includes facets, attributes, methods, and tag library, which can be used in multiple pages. Declarative components are reusable, composite UI components that are made up of other existing ADF Faces components. Suppose you are reusing the same components consistently in multiple circumstances. Instead of copying and pasting the commonly used UI elements repeatedly, you can define a declarative component that comprises those components, and then reuse that composite declarative component in multiple places or pages.

 **Note:**

If you want to use ADF Model layer bindings as values for the attributes, then you should use a page template instead. See [Using Page Templates](#).

To use declarative components in an application, you first create an XML-based declarative component definition, which is a JSF document written in XML syntax (with a file extension of `.jspx`). Declarative component JSF files do not contain the `f:view` and `document` tags, and they must have `componentDef` as the root tag.

The entire description of a declarative component is defined within two sections. One section is `xmlContent`, which contains all the page template component metadata that describes the declarative component's supported content areas. A declarative component's metadata includes the following:

- **Facets:** Facets act as placeholders for the content that will eventually be placed in the individual components that make up the declarative component. Each component references one facet. When page designers use a declarative component, they insert content into the facet, which in turn, allows the content to be inserted into the component.

 **Tip:**

Facets are the only area within a declarative component that can contain content. That is, when used on a JSF page, a declarative component may not have any children. Create facets for all areas where content may be needed.

- **Attributes:** You define attributes whose values can be used to populate attributes on the individual components. For example, if your declarative component uses a `panelBox` component, you may decide to create an attribute named `Title`. You may then design the declarative component so that the value of the `Title` attribute is used as the value for the `text` attribute of the `panelBox` component. You can provide default values for attributes that the user can then override.

 **Tip:**

Because users of a declarative component will not be able to directly set attributes on the individual components, you must be sure to create attributes for all attributes that you want users to be able to set or override the default value.

Additionally, if you want the declarative component to be able to use client-side attributes (for example, `attributeDragSource`), you must create that attribute and be sure to include it as a child to the appropriate component used in the declarative component. See [How to Create a Declarative Component](#).

- **Methods:** You can define a method to which you can bind a property on one of the included components. For example, if your declarative component contains a button, you can declare a method name and signature and then bind the `actionListener` attribute to the declared method. When page developers use the declarative component, they rebind to a method on a managed bean that contains the logic required by the component.

For example, say your declarative component contains a button that you knew always had to invoke an `actionEvent` method. You might create a declarative method named `method1` that used the signature `void method(javax.faces.event.ActionEvent)`. You might then bind the `actionListener` attribute on the button to the declared method. When page developers use the declarative component, JDeveloper will ask them to provide a method on a backing bean that uses the same signature.

- **Tag library:** All declarative components must be contained within a tag library that you import into the applications that will use them.

The second section (anything outside of the `xmlContent` tag) is where all the components that make up the declarative component are defined. Each component contains a reference back to the facet that will be used to add content to the component.

To use declarative components in a project, you first must deploy the library that contains the declarative component as an ADF Library. You can then add the deployed ADF Library JAR file to the project's properties, which automatically inserts the JSP tag library or libraries into the project's properties. Doing so allows the

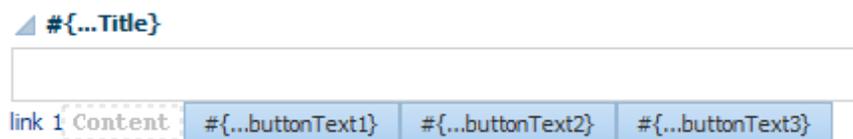
component(s) to be displayed in the Components window so that you can drag and drop them onto a JSF page.

For example, say you want to create a declarative component that uses a `panelBox` component. In the `panelBox` component's toolbar, you want to include three buttons that can be used to invoke `actionEvent` methods on a backing bean. To do this, create the following:

- One facet named `content` to hold the content of the `panelBox` component.
- One attribute named `Title` to determine the text to display as the `panelBox` component's title.
- Three attributes (one for each button, named `buttonText1`, `buttonText2`, and `buttonText3`) to determine the text to display on each button.
- Three attributes (one for each button, named `display1`, `display2`, `display3`) to determine whether or not the button will render, because you do not expect all three buttons will be needed every time the component is used.
- Three declarative methods (one for each button, named `method1`, `method2`, and `method3`) that each use the `actionEvent` method signature.
- One `panelBox` component whose `text` attribute is bound to the created `Title` attribute, and references the `content` facet.
- Three `Button` components. The `text` attribute for each would be bound to the corresponding `buttonText` attribute, the `render` attribute would be bound to the corresponding `display` attribute, and the `actionListener` attribute would be bound to the corresponding `method` name.

Figure 10-2 shows how such a declarative component would look in the visual editor.

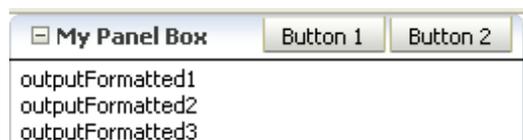
Figure 10-2 Declarative Component in the Visual Editor



When a page developer drops a declarative component that contains required attributes or methods onto the page, a dialog opens asking for values.

If the developer set values where only the first two buttons would render, and then added a `panelGroupLayout` component with output text, the page would render as shown in Figure 10-3.

Figure 10-3 Displayed Declarative Component



 **Note:**

You cannot use fragments or ADF databound components in the component layout of a declarative component. If you think some of the components will need to be bound to the ADF Model layer, then create attributes for those component attributes that need to be bound. The user of the declarative component can then manually bind those attributes to the ADF Model layer.

Additionally, because declarative components are delivered in external JAR files, the components cannot use the `jsp:include` tag because it will not be able to find the referenced files.

If your declarative component requires resources such as custom styles defined in CSS or JavaScript, then you need to include these using the `af:resource` tag on the consuming page. See [Adding Resources to Pages](#).

How to Create a Declarative Component

JDeveloper simplifies creating declarative component definitions by providing the Create ADF Declarative Component wizard, which lets you create facets, and define attributes and methods for the declarative component. The wizard also creates metadata in the `component-extension` tile that describes tag library information for the declarative component. The tag library metadata is used to create the JSP tag library for the declarative component.

First you add the template component metadata for facets and attributes inside the `xmlContent` section of the `componentDef` tag. After you have added all the necessary component metadata for facets and attributes, then you add the components that define the actual layout of the declarative component in the section outside of the `xmlContent` section.

 **Best Practice Tip:**

Because the tag library definition (TLD) for the declarative component must be generated before the component can be used, the component must be deployed to a JAR file before it can be consumed. It is best to create an application that contains only your declarative components. You can then deploy all the declarative components in a single library for use in multiple applications.

Before you begin:

It may be helpful to have an understanding of declarative components. See [Using Declarative Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To create a declarative component definition:

1. In the Applications window, right-click the node where you wish to create and store declarative components and click **New > From Gallery**.
2. In the New Gallery, expand **Web Tier**, select **JSF/Facelets**, then **ADF Declarative Component**, and click **OK**.
3. In the Create ADF Declarative Components dialog, enter a name and file name for the declarative component.

The name you specify will be used as the display name of the declarative component in the Components window, as well as the name of the Java class generated for the component tag. Only alphanumeric characters are allowed in the name for the declarative component, for example, `SampleName` or `myPanelBox`.

The file name is the name of the declarative component definition file (for example, `componentDef1.jspx`). By default, JDeveloper uses `.jspx` as the file extension because declarative component definition files must be XML documents.

4. Accept the default directory name for the declarative component, or choose a new location.

By default, JDeveloper saves declarative component definitions in the `/ViewController/public_html` directory in the file system. For example, you could save all declarative component definitions in the `/View Controller/public_html/declcomps` directory.

5. Enter a package name (for example, `dcomponent1`). JDeveloper uses the package name when creating the Java class for the declarative component.
6. Select a tag library to contain the new declarative component. If no tag library exists, or if you wish to create a new one, click **Add Tag Library**, and do the following to create metadata for the tag library:
 - a. Enter a name for the JSP tag library to contain the declarative component (for example, `dcompLib1`).
 - b. Enter the URI for the tag library (for example, `/dcomponentLib1`).
 - c. Enter a prefix to use for the tag library (for example, `dc`).
7. If you want to be able to add custom logic to your declarative component, select the **Use Custom Component Class** checkbox and enter a class name.
8. To add named facets, click the **Facet Definitions** tab and click the **Add** icon.

Facets in a declarative component are predefined areas where content can eventually be inserted. The components you use to create the declarative component will reference the facets. When page developers use the declarative components, they will place content into the facets, which in turn will allow the content to be placed into the individual components. Each facet must have a unique name. For example, your declarative component has a `panelBox` component, you could define a facet named `box-main` for the content area of the `panelBox` component.

9. To add attributes, click **Attributes** and click **Add**.

Attributes are UI component attributes that can be passed into a declarative component. Each attribute must have a name and class type (for example, `java.lang.String`). You can assign default values, and you can specify that the values are mandatory by selecting the **Required** checkbox.



Tip:

You must create attributes for any attributes on the included components for which you want users to be able to set or change values.

Remember to also add attributes for any tags you may need to add to support functionality of the component, for example values required by the `attributeDragSource` tag used for drag and drop functionality.

10. To add declarative methods, click the **Methods** tab and click the **Add** icon.

Declarative methods allow you to bind command component actions or action listeners to method signatures, which will later resolve to actual methods of the same signature on backing beans for the page on which the components are used. You can click the browse (...) icon to open the Method Signature dialog, which allows you to search for and build your signature.

When you complete the dialog, JDeveloper displays the declarative component definition file in the visual editor.



Tip:

Once a declarative component is created, you can add facets and attributes by selecting the `componentDef` tag in the Structure window, and using the Properties window.

11. In the Components window, drag and drop a component as a child to the `componentDef` tag in the Structure window.

Suppose you dropped a `panelBox` component. In the Structure window, JDeveloper adds the component after the `xmlContent` tag. It does not matter where you place the components for layout, before or after the `xmlContent` tag, but it is good practice to be consistent.

You can use any number of components in the component layout of a declarative component. Typically, you would add a component such as `panelFormLayout` or `panelGroupLayout`, and then add the components that define the layout into the panel component.

 **Note:**

You cannot use fragments or ADF databound components in the component layout of a declarative component. If you think some of the components will need to be bound to the ADF Model layer, then create attributes for those component attributes. The user of the declarative component can then manually bind those attributes to the ADF Model layer. For information about using the ADF Model layer, see Using ADF Model in a Fusion Web Application in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, because declarative components are delivered in external JAR files, the components cannot use the `jsp:include` tag because it will not be able to find the referenced files.

12. Within those components (in the layout section) where content can eventually be inserted by page authors using the component, use the `facetRef` tag to reference the appropriate named facet.

For example, if you have defined a `content` facet for the main content area, you might add the `facetRef` tag as a child in the `panelBox` component to reference the `content` facet. At design time, when the page developer drops components into the `content` facet, the components are placed in the `panelBox` component.

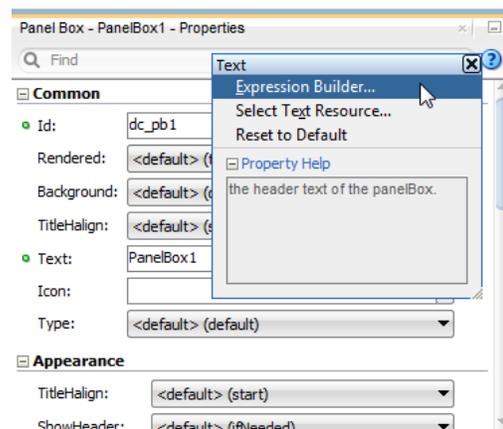
When you drag **Facet** from the Components window Core Structure panel and drop it in the desired location on the page, JDeveloper displays the Insert Facet Definition dialog. In that dialog, select a facet name from the dropdown list, or enter a facet name, and click **OK**. If you enter a facet name that is not already defined in the component metadata of the definition file, JDeveloper automatically adds an entry for the new facet definition in the component metadata within the `xmlContent` tag.

 **Note:**

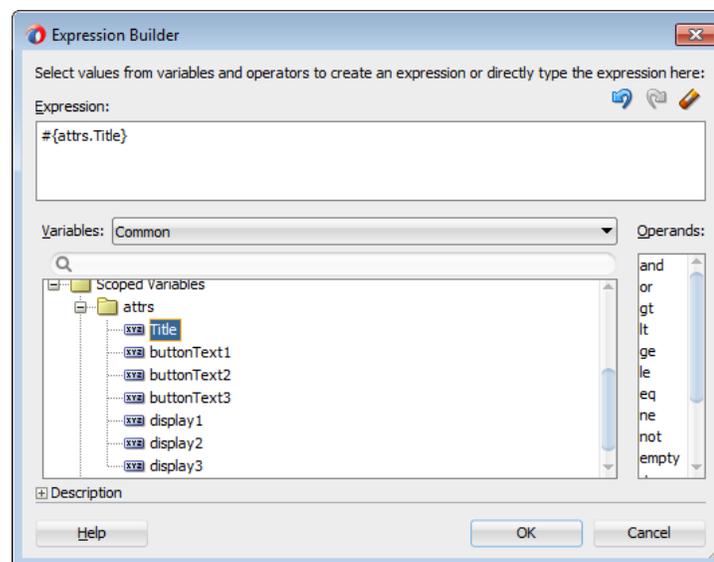
Each facet can be referenced only once. That is, you cannot use multiple `facetRef` tags referencing the same `facetName` value in the same declarative component definition.

13. To specify where attributes should be used in the declarative component, use the Properties window and the Expression Builder to bind component attribute values to the created attributes.

For example, if you have defined a `Title` attribute and added a `panelBox` as a component, you might use the dropdown menu next to the text attribute in the Properties window to open the Expression Builder, as shown in [Figure 10-4](#).

Figure 10-4 Opening the Expression Builder for an Attribute in the Properties Window

In the Expression Builder, you can expand the **Scoped Variables > attrs** node to select the created attribute that should be used for the value of the attribute in the Properties window. For example, [Figure 10-5](#) shows the **Title** attribute selected in the Expression Builder. Click **OK** to add the expression as the value for the attribute.

Figure 10-5 Expression Builder Displays Created Attributes

- To specify the methods that buttons in the declarative component should invoke, use the dropdown menu next to that component's `actionListener` attribute and choose **Edit** to open the Edit Property dialog. This dialog allows you to choose one of the declarative methods you created for the declarative component.

In the dialog, select **Declarative Component Methods**, select the declarative method from the dropdown list, and click **OK**.

What Happens When You Create a Declarative Component

When you first use the Create ADF Declarative Component wizard, JDeveloper creates the metadata file using the name you entered in the wizard. The entire definition for the component is contained in the `componentDef` tag. This tag uses two attributes. The first is `var`, which is a variable used by the individual components to access the attribute values. By default, the value of `var` is `attrs`. The second attribute is `componentVar`, which is a variable used by the individual components to access the methods. Be aware that if `componentVar` is set to `component` then it is incompatible with JSF 2.0 and the expression will fail. You will need to set `componentVar` using a different variable.

The metadata describing the facets, attributes, and methods is contained in the `xmlContent` tag. Facet information is contained within the `facet` tag, attribute information is contained within the `attribute` tag, and method information is contained within the `component-extension` tag, as is library information. The following example shows abbreviated code for the declarative component shown in [Figure 10-2](#).

```
<af:xmlContent>
  <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
    <display-name>myPanelBox</display-name>
    <facet>
      <description>Holds the content in the panel box</description>
      <facet-name>content</facet-name>
    </facet>
    <attribute>
      <attribute-name>Title</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
      <required>true</required>
    </attribute>
    <attribute>
      <attribute-name>buttonText1</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    . . .
    <component-extension>
      <component-tag-namespace>dcomponent1</component-tag-namespace>
      <component-taglib-uri>/componentLib1</component-taglib-uri>
      <method-attribute>
        <attribute-name>method1</attribute-name>
        <method-signature>
          void method( javax.faces.event.ActionEvent )
        </method-signature>
      </method-attribute>
      <method-attribute>
        <attribute-name>method2</attribute-name>
        <method-signature>
          void method( javax.faces.event.ActionEvent )
        </method-signature>
      </method-attribute>
    </component-extension>
  </component>
</af:xmlContent>
```

Metadata for the included components is contained after the `xmlContent` tag. The code for these components is the same as it might be in a standard JSF page, including any attribute values you set directly on the components. Any bindings you

created to the attributes or methods use the component's variables in the bindings. The following example shows the code for the `panelBox` component with the three buttons in the toolbar. Notice that the `facetRef` tag appears as a child to the `panelBox` component, as any content a page developer will add will then be a child to the `panelBox` component.

```
<af:panelBox text="{attrs.Title}" inlineStyle="width:25%;">
  <f:facet name="toolbar">
    <af:group>
      <af:toolbar>
        <af:button text="{attrs.buttonText1}"
                    actionListener="{component.handleMethod1}"
                    rendered="{attrs.display1}"/>
        <af:button text="{attrs.buttonText2}"
                    actionListener="{component.handleMethod2}"/>
        <af:button text="{attrs.buttonText3}"
                    actionListener="{component.handleMethod3}"/>
      </af:toolbar>
    </af:group>
  </f:facet>
  <af:facetRef facetName="content"/>
</af:panelBox>
```

The first time you use the wizard to create a declarative component in a project, JDeveloper automatically creates the `declarativecomp-metadata.xml` file, which is placed in the `/ViewController/src/META-INF` directory in the file system.

For each declarative component that you define using the wizard, JDeveloper creates a declarative component definition file (for example, `componentDef1.jspx`), and adds an entry to the `declarativecomp-metadata.xml` file. The following example shows an example of the `declarativecomp-metadata.xml` file.

```
<declarativeCompDefs
  xmlns="http://xmlns.oracle.com/adf/faces/rich/declarativecomp">
  <declarativecomp-jsp-ui-def>
    /componentDef1.jspx
  </declarativecomp-jsp-ui-def>
  <declarativecomp-taglib>
    <taglib-name>
      dcompLib1
    </taglib-name>
    <taglib-uri>
      /componentLib1
    </taglib-uri>
    <taglib-prefix>
      dc
    </taglib-prefix>
  </declarativecomp-taglib>
</declarativeCompDefs>
```

 **Note:**

When you rename or delete a declarative component in the Applications window, JDeveloper renames or deletes the declarative component definition file in the file system, but you must manually change or delete the declarative component entry in the `declarativecomp-metadata.xml` file, and update or remove any JSF pages that use the declarative component.

The `declarativecomp-metadata.xml` file contains the names, paths, and tag library information of all the declarative components you create in the project. When you deploy the project, the metadata is used by JDeveloper to create the JSP tag libraries and Java classes for the declarative components.

JDeveloper also adds entries to the `faces-config.xml` file.

How to Deploy Declarative Components

Declarative components require a tag library definition (TLD) in order to be displayed. JDeveloper automatically generates the TLD when you deploy the project. Because of this, you must first deploy the project that contains your declarative components before you can use them. This means before you can use declarative components in a project, or before you can share declarative components with other developers, you must deploy the declarative component definitions project to an ADF Library JAR file. For instructions on how to deploy a project to an ADF Library JAR file, see Reusing Application Components in *Developing Fusion Web Applications with Oracle Application Development Framework*.

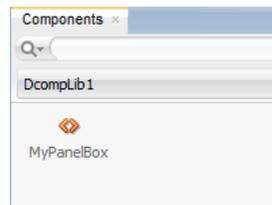
Briefly, when you deploy a project that contains declarative component definitions, JDeveloper adds the following for you to the ADF Library JAR file:

- A component tag class (for example, the `componentDef1Tag.class`) for each declarative component definition (that is, for each `componentDef` component)
- One or more JSP TLD files for the declarative components, using information from the project's `declarativecomp-metadata.xml` file

To use declarative components in a consuming project, you add the deployed ADF Library JAR file to the project's properties. For instructions on how to add an ADF Library JAR file, see Reusing Application Components in *Developing Fusion Web Applications with Oracle Application Development Framework*. By adding the deployed JAR file, JDeveloper automatically inserts the JSP tag library or libraries (which contain the reusable declarative components) into the project's properties, and also displays them in the Components window.

How to Use Declarative Components in JSF Pages

In JDeveloper, you add declarative components to a JSF page just like any other UI components, by selecting and dragging the components from the Components window, and dropping them into the desired locations on the page. Your declarative components appear in a page of the palette just for your tag library. [Figure 10-6](#) shows the page in the Components window for a library with a declarative component.

Figure 10-6 Components Window with a Declarative Component

When you drag a declarative component that contains required attributes onto a page, a dialog opens where you enter values for any defined attributes.

Once the declarative component is added to the page, you must manually bind the declarative methods to actual methods on managed beans.

Before proceeding with the following procedure, you must already have added the ADF Library JAR file that contains the declarative components to the project where you are creating JSF pages that are to consume the declarative components. For instructions on how to add an ADF Library JAR file, see *Reusing Application Components in Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of declarative components. See [Using Declarative Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To use declarative components in a JSF page:

1. In the Applications window, double-click the JSF page (or page template) to open it in the visual editor.
2. In the Components window, from the tag library panel, drag and drop the declarative component to the page.

You can add the same declarative component more than once on the same page.

If the declarative component definition contains any required attributes, JDeveloper opens a dialog for you to enter the required values for the declarative component that you are inserting.

3. Add components by dragging and dropping components from the Components window in the facets of the template. In the Structure window, expand the structure until you see the element for the declarative component, for example, `dc:myPanelBox`, where `dc` is the tag library prefix and `myPanelBox` is the declarative component name.

Under that are the facets (for example, `f:facet - content`) that have been defined in the declarative component definition. You add components to these facets.

You cannot add content directly into the declarative component; you can drop content into the named facets only. The types of components you can drop into a facet may be dependent on the location of the `facetRef` tag in the declarative component definition. For example, if you have defined `facetRef` to be a child of

table in the declarative component definition, then only column components can be dropped into the facet because table accepts column children only.

 **Note:**

You cannot place any components as direct children of a declarative component. All content to appear within a declarative component must be placed within a facet of that component.

4. In the Structure window, again select the declarative component element, for example, `dc:myPanelBox`. The Properties window displays all the attributes and methods that have been predefined in the declarative component definition (for example, `Title`). The attributes might have default values.

You can assign static values to the attributes, or you can use EL expressions (for example, `#{myBean.somevalue}`). For any of the methods, you must bind to a method that uses the same signature as the declared method defined on the declarative component.

At runtime, the attribute value will be displayed in the appropriate location as specified in the declarative component definition by the EL expression that bears the name of the attribute (for example, `#{attrs.someAttributeName}`).

5. If you need to include resources such as CSS or JavaScript, then you need to include these using the `af:resource` tag. See [Adding Resources to Pages](#).

What Happens When You Use a Declarative Component on a JSF Page

After adding a declarative component to the page, the visual editor displays the component's defined facets as named boxes, along with any content that is rendered by components defined in the component layout section of the declarative component definition.

Like other UI components, JDeveloper adds the declarative component tag library namespace and prefix to the `jsp:root` tag in the page when you first add a declarative component to a page, for example:

```
<jsp:root xmlns:dc="/dcomponentLib1: ..>
```

In this example, `dc` is the tag library prefix, and `/dcomponentLib1` is the namespace.

JDeveloper adds the tag for the declarative component onto the page. The tag includes values for the component's attributes as set in the dialog when adding the component. The following example shows the code for the `MyPanelBox` declarative component to which a user has added a `panelGroupLayout` component that contains three `outputFormatted` components.

```
<dc:myPanelBox title="My Panel Box" buttonText1="Button 1"
    display1="true" display2="true" buttonText2="Button 2"
    display3="false">
  <f:facet name="Content">
    <af:panelGroupLayout layout="scroll">
      <af:outputFormatted value="outputFormatted1"
        styleUsage="instruction"/>
```

```
<af:outputFormatted value="outputFormatted2"
                    styleUsage="instruction" />
<af:outputFormatted value="outputFormatted3"
                    styleUsage="instruction" />
</af:panelGroupLayout>
</f:facet>
</dc:myPanelBox>
```

What Happens at Runtime: Declarative Components

When a JSF page that consumes a declarative component is executed:

- The declarative component tag in the consuming page locates the declarative component tag class and definition file that contains the declarative component metadata and layout.
- The component subtree defined in the layout section of the `componentDef` tag is instantiated and inserted into the consuming page's component tree at the location identified by the declarative component tag in the page.
- The `componentDef` tag sets the value of the `var` attribute so that the declarative component can internally reference its own attributes. The declarative component just sets the attribute values; the runtime maps those values into the attributes defined in the `componentDef` tag.
- Using declarative component metadata, the declarative component applies any default values to its attributes and checks for required values.
- The consuming page passes facet contents into the declarative component by using the `facet` tag. The facet contents of each `facet` tag are inserted into the appropriate location on the declarative component as specified by the corresponding `facetRef` tag in the layout section of the `componentDef` tag.

Adding Resources to Pages

For reusable contents that include ADF Faces facets, attributes, and methods, you must use the `af:resource` tag to define the location of the resource. Without the resource tag, the resources won't get added to a page. You should use the `af:resource` tag to add CSS or JavaScript to pages, page templates, or declarative components. This tag is especially useful for page templates and declarative components because resources can only be added to the page (in the HTML head element). When you can use this tag in page templates and declarative components, the resources will be added to the consuming page during JSP execution. If this tag is not used, browsers may need to re-layout pages that use page templates and declarative components whenever it encounters a style or link tag. The resources can be added to the page during any page request, but they must be added before the document component is rendered.

The resource tag can be used with PPR. During PPR, the following requirements apply:

- URL resources are compared on the client before being added to the page. This ensures duplicates are not added.
- CSS resources are removed from the page during a PPR navigation. The new page will have the new CSS resources.

- The resource used in the `af:resource` tag is not the ADF skin CSS. For information on using the ADF skin, see [Customizing the Appearance Using Styles and Skins](#).

How to Add Resources to Page Templates and Declarative Components

You use the `af:resource` tag to define the location of the resource. The resource will then be added to the document header of the consuming page.

Before you begin:

It may be helpful to have an understanding of the available resources. See [Adding Resources to Pages](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Reusable Components](#).

To add resources:

1. In the Components window, from the Layout panel, in the Core Structure group, drag and drop a **Resource** to the page.
2. In the Insert Resource dialog, do the following:
 - **Type:** Select `css` or `javascript` from the dropdown list.
 - **Resource:** Enter the URI of the source of the external resource to include in the page. If the URI starts with a slash (`/`), the URI will be made context relative, if it starts with two slashes (`//`), it will be made server relative. All others are non-absolute URIs are relative to URI location in the browser. If not given, the resource content will be taken from the tag body.
3. Click **OK**.

What Happens at Runtime: How to Add Resources to the Document Header

During JSP tag execution, the `af:resource` tag only executes if its parent component has been created. When it executes, it adds objects to a set in the RichDocument component. RichDocument then adds the specified resources (CSS or JavaScript) to the consuming page.

Part IV

Using Common ADF Faces Components

Part IV documents the different ADF Faces components, including input, collection, lists, queries, menus, dialogs, calendars, output, messages, navigation, and dynamic components.

Specifically, this part contains the following chapters:

- [Using Input Components and Defining Forms](#)
- [Using Tables, Trees, and Other Collection-Based Components](#)
- [Using List-of-Values Components](#)
- [Using Query Components](#)
- [Using Menus, Toolbars, and Toolboxes](#)
- [Using Popup Dialogs, Menus, and Windows](#)
- [Using a Calendar Component](#)
- [Using Output Components](#)
- [Displaying Tips, Messages, and Help](#)
- [Working with Navigation Components](#)
- [Determining Components at Runtime](#)

11

Using Input Components and Defining Forms

This chapter describes how to define forms and create input components that allow end users to enter data (such as `inputText`), select values (such as `inputNumber`, `inputRange`, `inputColor`, `inputDate`, and `select` components), edit text (such as `richTextEditor`), and load files (such as `inputFile`).

This chapter includes the following sections:

- [About Input Components and Forms](#)
- [Defining Forms](#)
- [Using the `inputText` Component](#)
- [Using the Input Number Components](#)
- [Using Color and Date Choosers](#)
- [Using Selection Components](#)
- [Using Shuttle Components](#)
- [Using the `richTextEditor` Component](#)
- [Using File Upload](#)
- [Using Code Editor](#)

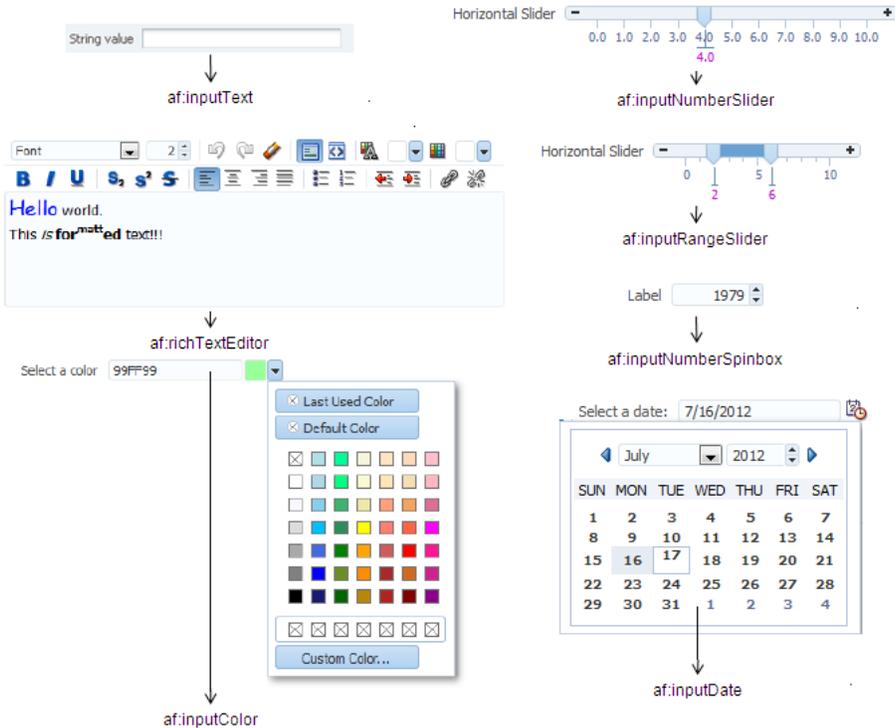
If you want to use an input component that selects from a list of items that may be potentially large, or may represent relationships between objects (such as creating a list to represent an attribute that is a foreign key to another object), then you may want to use a list of values component. For information about those components, see [Using List-of-Values Components](#).

About Input Components and Forms

The ADF input components is a means by which input data can be submitted to the application. You can use various input components in your forms that provides built-in functions to collect input data from the users, such as creating a new file, adding a rich text editor to the form, or providing a multiple choice list.

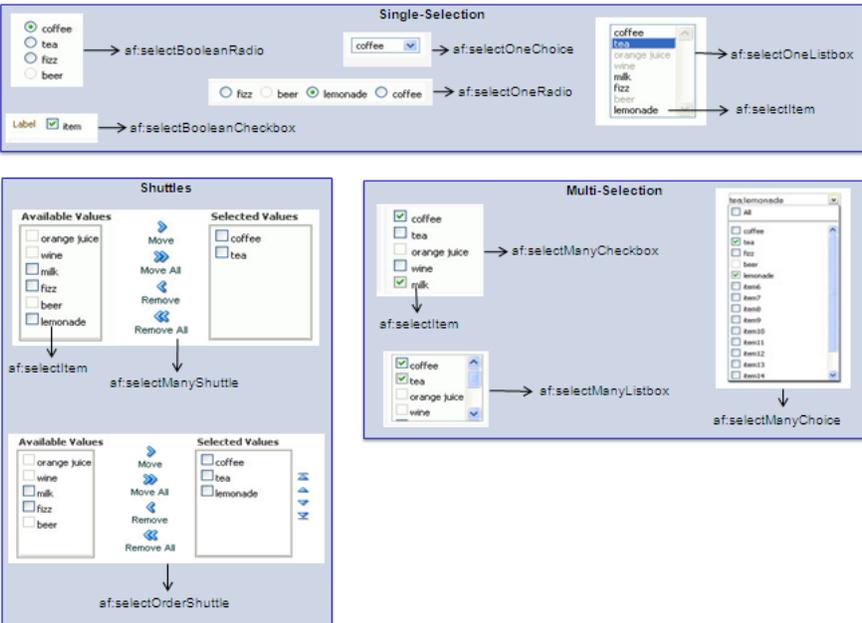
Input components accept user input in a variety of formats. The most common formats are text, numbers, date, and selection lists that appear inside a form and are submitted when the form is submitted. The entered values or selections may be validated and converted before they are processed further. [Figure 11-1](#) shows ADF Faces standard input components.

Figure 11-1 ADF Faces Input Components



ADF Faces input components also include a number of components that allow users to select one or multiple values, as shown in Figure 11-2.

Figure 11-2 Select Components



Input Component Use Cases and Examples

Input components are often used to build forms for user input. For example, the File Explorer application contains a form that allows users to create a new file. As shown in [Figure 11-3](#), input components allow users to enter the name, the size, select permissions, and add keywords, and a description for a file. The Name field is required, as noted by the asterisk. If a user fails to enter a value, an error message is displayed. That validation and associated error message may be configured on the component (by setting the `required` or `requiredMessageDetail` attribute), or handled on the server (by setting the `showRequired` attribute).

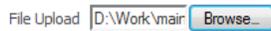
Figure 11-3 Form Uses Input Components

The `richTextEditor` component provides rich text input that can span many lines and can be formatted using different fonts, sizes, justification, and other editing features that may be required when you want users to enter more than simple text. For example, the `richTextEditor` might be used in a web-based discussion forum, allowing users to format the text that they need to publish, as shown in [Figure 11-4](#).

Figure 11-4 richTextEditor Used in a Discussion Forum

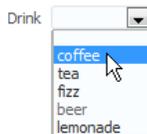
The `inputFile` component allows users to browse for a local file to upload to the application server. For example, an email message might allow users to attach a file to a message, as shown in [Figure 11-5](#).

Figure 11-5 fileUpload Component



The ADF Faces selection components allows users to make selections from a list of items instead of typing in values. ADF Faces provides both single choice selection lists and multi-choice selection lists. Single-choice lists are used to select one value from a list, such as the desired drink in an online food order, as shown in [Figure 11-6](#).

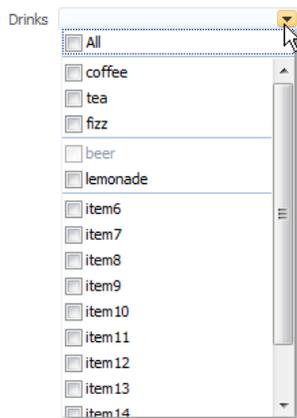
Figure 11-6 Users Can Select One Value From the selectOneChoice Component



ADF single-selection components include a dropdown list (as shown in [Figure 11-6](#)), a list box, radio buttons, and checkboxes.

ADF multi-selection components allow users to select more than one value in a list. For example, instead of being able to select just one drink type, the `selectManyChoice` component allows a user to select more than one drink, as shown in [Figure 11-7](#).

Figure 11-7 Users Can Select Multiple Values From the selectManyChoice Component



ADF multiple choice components include a dropdown list, checkboxes, a checkbox list, a shuttle, and an ordered shuttle.

 **Best Practice:**

You can use either selection lists or list-of-values (LOV) components to display a list. LOV components should be used when the selection list is large. LOV components are model-driven using the `ListOfValueModel` class and may be configured programmatically using the API. They present their selection list inside a popup window that may also include a query panel. Selection lists simply display a static list of values. LOV components are single select and allow you to select only one option, hence should not be used for multiple selections. For information about using LOV components, see [Using List-of-Values Components](#).

The form components provide a container for other components. The `form` component represents a region where values from embedded input components can be submitted. Only one form component per page is supported. ADF Faces also provides the `subform` component, which adds flexibility by defining subregions whose component values can be submitted separately within a form.

 **Note:**

If you are using an input component that is configured to display a list of suggestions with the `af:autoSuggestBehavior` tag and uses `AdfCustomEvent` to send custom event to the server, the server listener will not be executed as `af:autoSuggestBehavior` tag interrupts the input value life cycle after Apply request value phase and jumps to render response phase.

Additional Functionality for Input Components and Forms

You may find it helpful to understand other ADF Faces features before you implement your input components. Additionally, once you have added an input component or form to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that input components can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Client components:** Input components can be client components. To work with the components on the client, see [Using ADF Faces Client-Side Architecture](#).
- **JavaScript APIs:** All input components have JavaScript client APIs that you can use to set or get property values. See the *JavaScript API Reference for Oracle ADF Faces*.
- **Events:** Input components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- You can add validation and conversion to input components. See [Validating and Converting Input](#).

- You can display tips and messages, as well as associate online help with input components. See [Displaying Tips, Messages, and Help](#).
- There may be times when you want the certain input components to be validated before other components on the page. See [Using the Immediate Attribute](#).
- You may want other components on the page to update based on selections you make from a selection component. See [Using the Optimized Lifecycle](#).
- You may want to use the `scrollComponentIntoViewBehavior` tag with the `richTextEditor` component to allow users to jump to specific areas in the component. See [How to Use the scrollComponentIntoViewBehavior Tag](#).
- You can change the icons used for required and changed notifications using skins. See [Customizing the Appearance Using Styles and Skins](#).
- You can make your input components accessible. See [Developing Accessible ADF Faces Pages](#).
- Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- If your application uses ADF Model, then you can create automatically bound forms using data controls (whether based on ADF Business Components or other business services). See [Creating a Basic Databound Page in Developing Fusion Web Applications with Oracle Application Development Framework](#).

Defining Forms

The ADF shuttle component that you add to a page lets users make a single selection or multiple selections from an available list of values box and add them to a selected list of values box.

A **form** is a component that serves as a container for other components. When a submit action occurs within the form, any modified input values are submitted. For example, you can create an input form that consists of input and selection components, and a submit command button, all enclosed within a form. When the user enters values into the various input fields and clicks the Submit button, those new input values will be sent for processing.

By default, when you create a JSF page in JDeveloper, it automatically inserts a `form` component into the page. When you add components to the page, they will be inserted inside the `form` component.

Tip:

If you do not already have an `af:form` tag on the page, and you drag and drop an ADF Faces component onto the page, JDeveloper will prompt you to enclose the component within a form component.

The following example shows two input components and a Submit button that when clicked will submit both input values for processing.

```
<af:form id="f1">
  <af:panelFormLayout id="pf11">
    <af:inputText value="#{myBean.firstName}"
```

```

        label="#{FirstName}"
        id="it1">
</af:inputText>
<af:inputText value="#{myBean.lastName}"
        label="#{LastName}"
        id="it2">
</af:inputText>
<af:button text="Submit" id="b1"/>
</af:panelFormLayout>
</af:form>

```

Because there can be only one `form` component on a page, you can use subforms within a form to create separate regions whose input values can be submitted. Within a region, the values in the subform will be validated and processed only if a component inside the subform caused the values to be submitted. You can also nest a subform within another subform to create nested regions whose values can be submitted. For information about subforms, see [Using Subforms to Create Sections on a Page](#).

The following example shows a form with two subforms, each containing its own input components and buttons, such as the Submit button. When a Submit button is clicked, only the input values within that subform will be submitted for processing.

```

<af:form id="f1">
  <af:subform id="s1">
    <af:panelFormLayout id="pfl1">
      <af:inputText value="#{myBean.firstName}"
        label="#{FirstName}"
        id="it1">
    </af:inputText>
    <af:inputText value="#{myBean.lastName}"
        label="#{LastName}"
        id="it2">
    </af:inputText>
    <af:button text="Submit" id="b1"/>
    </af:panelFormLayout>
  </af:subform>

  <af:subform id="s2">
    <af:panelFormLayout id="pfl2">
      <af:inputText value="#{myBean.primaryPhone}"
        label="#{PrimaryPhone}"
        id="it3">
    </af:inputText>
    <af:inputText value="#{myBean.cellPhone}"
        label="#{CellPhone}"
        id="it4">
    </af:inputText>
    <af:button text="Submit" id="b2"/>
    </af:panelFormLayout>
  </af:subform>
</af:form>

```

Aside from the basic button, you can add any other command component within a form and have it operate on any field within the form.

How to Add a Form to a Page

In most cases, JDeveloper will add the form component for you. However, there may be cases where you must manually add a form, or configure the form with certain attribute values.

Before you begin:

It may be helpful to have an understanding of form components. See [Defining Forms](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add a form to a page:

1. In the Components window, from the Layout panel, in the Core Structure group, drag and drop a **Form** onto the page.
2. In the Properties window, expand the Common section, where you can optionally set the following:
 - **DefaultCommand:** Specify the ID attribute of the command component whose action should be invoked when the **Enter** key is pressed and the focus is inside the form.
 - **UsesUpload:** Specify whether or not the form supports uploading files. The default is `False`. For information about uploading files, see [Using File Upload](#).
 - **TargetFrame:** Specify where the new page should be displayed. Acceptable values are any of the valid values for the target attribute in HTML. The default is `_self`.

How to Add a Subform to a Page

You should add subform components within a form component when you need a section of the page to be capable of independently submitting values.

Before you begin:

It may be helpful to have an understanding of forms and subforms. See [Defining Forms](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

You must add a form component to the page. For procedures, see [How to Add a Form to a Page](#).

To add subforms to a page:

1. In the Components window, from the Layout panel, in the Core Structure group, drag and drop a **Subform** onto the page as a child to a `form` component.
2. In the Properties window, expand the Common section and set the following:
 - **Default:** Specify whether or not the subform should assume it has submitted its values. When set to the default value of `false`, this `subform` component will consider itself to be submitted only if no other `subform` component has been submitted. When set to `true`, this subform component assumes it has submitted its values.

 **Tip:**

A `subform` is considered submitted if an event is queued by one of its children or facets for a phase later than Apply Request Values (that is, for later than `decode()`). For information about lifecycle phases, see [Using the JSF Lifecycle with ADF Faces](#).

- **Default Command:** Specify the ID attribute of the command component whose action should be invoked when the Enter key is pressed and the focus is inside the subform.

How to Add a Button to Reset the Form

You add the button component and configure it using `af:resetListener` to reset other input components to their default values. The reset button will act upon only those components within that form or subform.

Before you begin:

It may be helpful to have an understanding of form components. See [Defining Forms](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add a button with `resetListener` to the page:

1. In the Components window, from the General Controls panel, drag and drop a **Button** onto the page.
2. In the Properties window, set the following:
 - **Text:** Specify the textual label of the button.
The property is available in the Common section.
 - **Disabled:** Specify whether or not the button should be disabled. For example, you could enter an EL expression that determines certain conditions under which the button should be disabled.
The property is available in the Behavior section.
3. In the Components window, from the Operations panel, in the Listeners group, drag a **Reset Listener** component and drop it as a child to the button component.
4. In the Insert Reset Listener dialog, select `action` from the **Type** dropdown, and click **OK**.

Using the `inputText` Component

The ADF `inputText` component is a textual input field. You can allow users to submit textual input data using these components. You can allow users to enter a single row input text or multiple rows of input text.

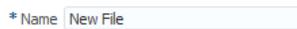
Although input components include many variations, such as pickers, sliders, and a spinbox, the `inputText` component is the basic input component for entering values. You can define an `inputText` component as a single-row input field or as a text area by setting the `rows` attribute to more than 1. However, if you want user to enter rich

text, consider using the `richTextEditor` component as described in [Using the richTextEditor Component](#).

You can allow auto-completion for an `inputText` component using the `autoComplete` attribute. When set to `true`, the component remembers previous entries, and then displays those entries when the user types in values that begin to match those entries.

You can hide the input values from being displayed, such as for passwords, by setting the `secret` attribute to `true`. Like other ADF Faces components, the `inputText` component supports label, text, and messages. When you want this component to be displayed without a label, you set the `simple` attribute to `true`. [Figure 11-8](#) shows a single-row `inputText` component.

Figure 11-8 Single-Row `inputText` Component

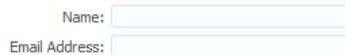


* Name

You can make the `inputText` component display more than one row of text using the `rows` attribute. If you set the `rows` attribute to be greater than one, and you set the `simple` attribute to `true`, then the `inputText` component can be configured to stretch to fit its container using the `dimensionsFrom` attribute. For information about how components stretch, see [Geometry Management and Component Stretching](#). [Figure 11-10](#) shows a multi-row `inputText` component.

You can add multiple `inputText` components to create an input form. [Figure 11-9](#) shows an input form using two `inputText` components.

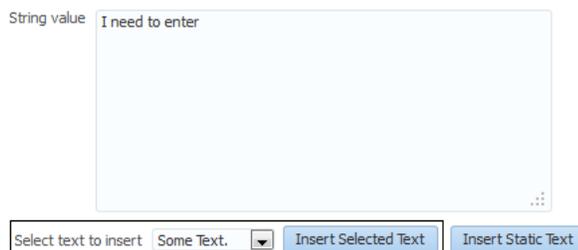
Figure 11-9 Form Created by `inputText` Components



Name:
Email Address:

You can also configure an `insertTextBehavior` tag that works with command components to insert given text into an `inputText` component. The text to be entered can be a simple string, or it can be the value of another component, for example the selected list item in a `selectOneChoice` component. For example, [Figure 11-10](#) shows an `inputText` component with some text already entered by a user.

Figure 11-10 `inputText` Component with Entered Text

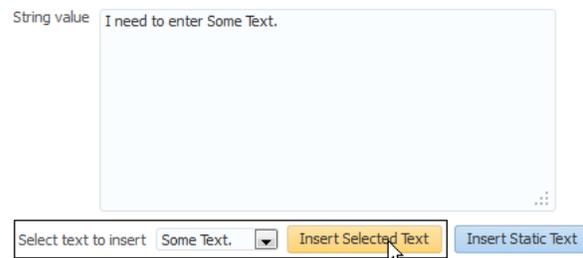


String value

Select text to insert

The user can then select additional text from a dropdown list, click the command button, and that text appears in the `inputText` component as shown in [Figure 11-11](#).

Figure 11-11 `inputText` Component with Inserted Text



How to Add an `inputText` Component

You can use an `inputText` component inside any of the layout components described in [Organizing Content on Web Pages](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using the `inputText` Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add an `inputText` component:

1. In the Components window, from the Text and Selection panel, drag and drop an **Input Text** onto the page.
2. In the Properties window, expand the Common section and set the following:
 - **Label:** Enter a value to specify the text to be used as the label.
If the text to be used is stored in a resource bundle, use the dropdown list to select **Select Text Resource**. Use the Select Text Resource dialog either to search for appropriate text in an existing bundle, or to create a new entry in an existing bundle. For information about using resource bundles, see [Internationalizing and Localizing Pages](#).
 - **Value:** Specify the value of the component. If the EL binding for a value points to a bean property with a `get` method but no `set` method, and this is a component whose value can be edited, then the component will be rendered in read-only mode.

Note:

If you are using an `inputText` component to display a Character Large Object (CLOB), then you will need to create a custom converter that converts the CLOB to a String. For information about conversion, see [Creating Custom ADF Faces Converters](#).

3. Expand the Appearance section, and set the following:
 - **Columns:** Specify the number of visible characters in the text field.
 - **Rows:** Specify the height of the text control by entering the number of rows to be shown. The default value is 1, which generates a one-row input field. The number of rows is estimated based on the default font size of the browser.

If you want to change the default text wrapping behavior when **Rows** is set to more than 1, you must also set the `wrap` attribute.
 - **DimensionsFrom:** Determine how you want the `inputText` component to handle geometry management. Set this attribute to one of the following:
 - `auto`: If the parent component to the `inputText` component allows stretching of its child, then the `inputText` component will stretch to fill the parent component, as long as the `rows` attribute is set to a number greater than one and the `simple` attribute is set to `true`. If the parent component does not allow stretching, then the `inputText` component gets its dimensions from the content.
 - `content`: The `inputText` component gets its dimensions from the component content. This is the default.
 - `parent`: The `inputText` component gets its dimensions from the `inlineStyle` attribute. If no value exists for `inlineStyle`, then the size is determined by the parent container.
 - **Secret:** Specify this boolean value that applies only to single-line text controls. When set to `true`, the `secret` attribute hides the actual value of the text from the user.
 - **Wrap:** Specify the type of text wrapping to be used in a multiple-row text control. This attribute is ignored for a single-row component. By default, the attribute is set to `soft`, which means multiple-row text wraps visually, but does not include carriage returns in the submitted value. Setting this attribute to `off` will disable wrapping: the multiple-row text will scroll horizontally. Setting it to `hard` specifies that the value of the text should include any carriage returns needed to wrap the lines.
 - **ShowRequired:** Specify whether or not to show a visual indication that the field is required. Note that setting the `required` attribute to `true` will also show the visual indication. You may want to use the `showRequired` attribute when a field is required *only* if another field's value is changed.
 - **Changed:** Specify whether or not to show a blue circle whenever the value of the field has changed. If you set this to `true`, you may also want to set the `changedDesc` attribute.
 - **ChangedDesc:** Specify the text to be displayed in a tooltip on a mouseover of the changed icon. By default, the text is "Changed." You can override this by providing a different value.
 - **Editable:** Determine whether you want the component to always appear editable. If so, select `always`. If you want the value to appear as read-only until the user hovers over it, select `onAccess`. If you want the value to be inherited from an ancestor component, select `inherit`.

 **Note:**

If you select `inherit`, and no ancestor components define the `editable` value, then the value always is used.

- **AccessKey:** Specify the key to press that will access the field.
- **LabelAndAccessKey:** Instead of specifying a separate label and access key, you can combine the two, so that the access key is part of the label. Simply precede the letter to be used as an access key with an ampersand (&).

For example, if the label of a field is **Description** and you want the **D** to be the access key, you would enter `&Description`.

 **Note:**

Because the value is being stored in the source of the page in XML, the ampersand (&) character must be escaped, so the value will actually be represented in the source of the page using the characters `&` to represent the ampersand.

- **Simple:** Set to `true` if you do not want the label to be displayed.
- **Placeholder:** Specify the text that appears in the input component if the component is empty and does not have focus. When the component gets focus, or has a value, then the placeholder text is hidden.

The placeholder text is used to inform the user what should be entered in the input component.

 **Note:**

The placeholder value works on browsers that fully support HTML5.

4. If you want to style the label text, expand the **Style** section and set **LabelStyle**. Enter a CSS style property and value. For example, if you do not want the label text to wrap, you would enter `white-space:nowrap;` for the value
5. Expand the **Behavior** section and set the following:
 - **Required:** Specify whether or not a value is required. If set to `true`, a visual indication is displayed to let the user know a value must be entered. If a value is not entered, an exception will occur and the component will fail validation.
 - **ReadOnly:** Specify whether the control is displayed as a field whose value can be edited, or as an output-style text control.
 - **AutoSubmit:** Specify whether or not the component will automatically submit when the value changes. For information about using the `autoSubmit` attribute, see [Using the Optimized Lifecycle](#).
 - **AutoComplete:** Set to `on` to allow the component to display previous values when the user begins to enter a matching value. Set to `off` if no matches should be displayed. Default is `on`.

- **AutoTab:** Specify whether or not focus will automatically move to the next tab stop when the maximum length for the current component is reached.
 - **Usage:** Specify how the input component will be rendered in HTML 5 browser. The valid values are `auto`, `text`, and `search`. Default is `auto`.

If the usage type is `search`, the input component will render as an HTML 5 search input type. Some HTML 5 browsers may add a **Cancel** icon that can be used to clear the search text.
 - **MaximumLength:** Specify the maximum number of characters per line that can be entered into the text control. This includes the characters representing the new line. If set to 0 or less, the `maximumLength` attribute is ignored. Note that in some browsers such as Internet Explorer, a new line is treated as two characters.
 - **Converter:** Specify a converter object. See [Adding Conversion](#).
 - **Validator:** Specify a method reference to a validator method using an EL expression. See [Adding Validation](#).
6. If you want to disable spellcheck, expand the Other section and set **SpellCheck** to `off`. By default, the spellcheck is set to the spellcheck setting of the browser.
- For example, in Mozilla Firefox, the spellcheck feature is enabled by default.

How to Add the Ability to Insert Text into an `inputText` Component

The `insertTextBehavior` tag works with command components to insert given text into an `inputText` component. The text to be entered can be a simple string, or it can be the value of another component, for example the selected list item in a `selectOneChoice` component. To allow text to be inserted into an `inputText` component, add the `insertTextBehavior` tag as a child to a command component that will be used to insert the text.

Note:

The `insertTextBehavior` tag cancels server-side event delivery automatically; `actionListener` or `action` attributes on the parent command component will be ignored. If you need to also trigger server-side functionality, you must add an custom client listener to deliver the server-side event. See [Sending Custom Events from the Client to the Server](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using the `inputText` Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

Before you add an `insertTextBehavior` tag, you need to create an `inputText` component as described in [How to Add an `inputText` Component](#). Set the `clientComponent` attribute to `true`.

To add text insert behavior:

1. Add a command component that the user will click to insert the text. For procedures, see [How to Use Buttons and Links for Navigation and Deliver ActionEvents](#).
2. In the Components window, from the Operations panel, in the Behavior group, drag and drop an **Insert Text Behavior** as a child to the command component.
3. In the Insert Text Behavior dialog, enter the following:
 - **For:** Use the dropdown arrow to select **Edit** and then navigate to select the `inputText` component into which the text will be inserted.
 - **Value:** Enter the value for the text to be inserted. If you want to insert static text, then enter that text. If you want the user to be able to insert the value of another component (for example, the value of a `selectOneChoice` component), then enter an EL expression that resolves to that value. The following example shows page code for an `inputText` component into which either the value of a dropdown list or the value of static text can be inserted.

```
<af:inputText clientComponent="true"
              id="idInputText"
              label="String value"
              value="#{demoInput.value}"
              rows="10"
              columns="60">
</af:inputText>
<af:selectOneChoice id="targetChoice"
                  autoSubmit="true"
                  value="#{demoInput.choiceInsertText}"
                  label="Select text to insert">
  <af:selectItem label="Some Text." value="Some Text." id="si1"/>
  <af:selectItem label="0123456789" value="0123456789" id="si2"/>
  <af:selectItem label="~!@#%*^" value="~!@#%*^" id="si3"/>
  <af:selectItem label="Two Lines" value="\\nLine 1\\nLine 2" id="si4"/>
</af:selectOneChoice>
<af:button text="Insert Selected Text"
           id="firstButton"
           partialTriggers="targetChoice">
  <af:insertTextBehavior for="idInputText"
                       value="#{demoInput.choiceInsertText}">
  </af:insertTextBehavior>
</af:button>
<af:button text="Insert Static Text" id="b1">
  <af:insertTextBehavior for="idInputText" value="Some Static Text."/>
</af:button>
```

4. By default, the text will be inserted when the action event is triggered by clicking the command component. However, you can change this to another client event by choosing that event from the dropdown menu for the `triggerType` attribute of the `insertTextBehavior` component in the Properties window.

Using the Input Number Components

An ADF input number component is a numerical input field. You can allow users to submit numerical input data using these components. Some of the number components are `inputNumberSlider` in Horizontal and Vertical layout, `inputRangeSlider`, and `inputNumberSpinbox` components.

The slider components present the user with a slider with one or two markers whose position on the slider corresponds to a value. The slider values are displayed and include a minus icon at one end and a plus icon at the other. The user selects the marker and moves it along the slider to select a value. The `inputNumberSlider` component has one marker and allows the user to select one value from the slider, as shown in [Figure 11-12](#) in horizontal layout, and in [Figure 11-13](#) in vertical layout.

Figure 11-12 `inputNumberSlider` in Horizontal Layout

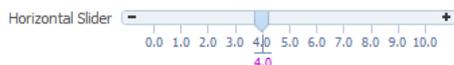
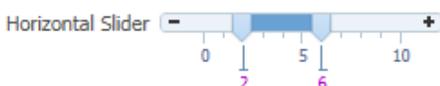


Figure 11-13 `inputNumberSlider` in Vertical Layout



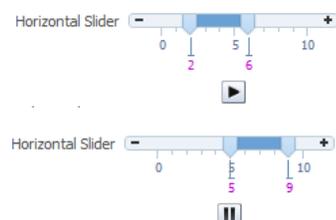
The `inputRangeSlider` component has two markers and allows the user to pick the end points of a range, as shown in [Figure 11-14](#).

Figure 11-14 `inputRangeSlider` in horizontal layout



You can also configure the `inputNumberSlider` and `inputRangeSlider` components to add a play/pause button that animates the slider across the component's increment values, as shown in [Figure 11-15](#).

Figure 11-15 `inputRangeSlider` with Play/Pause Button



The `inputNumberSpinbox` is an input component that presents the user with an input field for numerical values and a set of up- and down-arrow keys to increment or decrement the current value in the input field, as shown in [Figure 11-16](#).

Figure 11-16 `inputNumberSpinbox`



Label

How to Add an `inputNumberSlider` or an `inputRangeSlider` Component

When you add an `inputNumberSlider` or an `inputRangeSlider` component, you can determine the range of numbers shown and the increment of the displayed numbers.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using the Input Number Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add an `inputNumberSlider` or `inputRangeSlider` component:

1. In the Components window, from the Text and Selection panel, drag and drop a **Slider (Number)** or **Slider (Range)** onto the page.
2. In the Properties window, expand the Common section (and for the `inputRangeSlider` component, also expand the **Data** section) and set the following attributes:
 - **Label:** Specify a label for the component.
 - **Minimum:** Specify the minimum value that can be selected. This value is the begin value of the slider.
 - **Maximum:** Specify the maximum value that can be selected. This value is the end value of the slider.
 - **MinimumIncrement:** Specify the smallest possible increment. This is the increment that will be applied when the user clicks the plus or minus icon.
 - **MajorIncrement:** Specify the distance between two major marks. This value causes a labeled value to be displayed. For example, the `majorIncrement` value of the `inputRangeSlider` component in [Figure 11-14](#) is 5.0. If set to less than 0, major increments will not be shown.
 - **MinorIncrement:** Specify the distance between two minor marks. If less than 0, minor increments will not be shown.
 - **Value:** Specify the value of the component. If the EL binding for `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.
3. Expand the Appearance section and set the **Orientation** to specify whether the component will be in horizontal or vertical layout. For information about the other attributes in this section, see [How to Add an `inputText` Component](#).
4. Expand the Other section and set **AnimationInterval** to a value in milliseconds. Default value is zero.

If the value is greater than zero, a play button appears below the component. When clicked, it animates the slider across its increment values, stopping at each increment for the specified number of milliseconds. While animation is playing, the play button changes to a pause button that stops the animation at the current increment value.

For example, the `animationInterval` value of the `inputRangeSlider` component in [Figure 11-15](#) is 999.

How to Add an `inputNumberSpinbox` Component

The `inputNumberSpinbox` component allows the user to scroll through a set of numbers to select a value.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using the Input Number Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

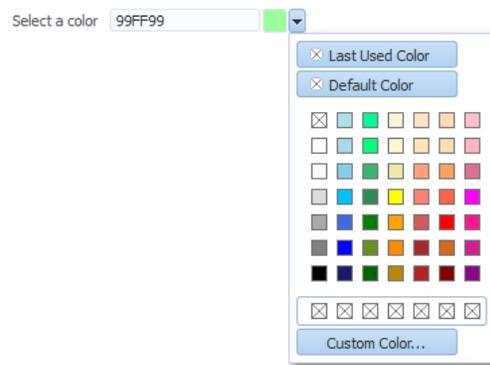
To add an `inputNumberSpinbox` component:

1. In the Components window, from the Text and Selection panel, drag and drop an **Input Number Spinbox** onto the page.
2. In the Properties window, expand the Data section, and set the following:
 - **Value:** Specify the value of the component. If the EL binding for `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.
 - **Minimum:** Specify the minimum value allowed in the input field.
 - **Maximum:** Specify the maximum value allowed in the input field.
 - **StepSize:** Specify the increment by which the spinbox will increase or decrease the number in the input field.
3. Expand the Appearance and Behavior sections and set the attributes. For information about setting these attributes, see [How to Add an `inputText` Component](#).

Using Color and Date Choosers

Using the ADF `inputColor` and `inputDate` component you can add a color pallet and a popup calendar in your form.

The `inputColor` component allows users to pick a color from a palette. It presents a text input field for entering code for colors. It also displays a button for picking colors from a palette in a popup, as shown in [Figure 11-17](#).

Figure 11-17 inputColor Component with Popup chooseColor Component

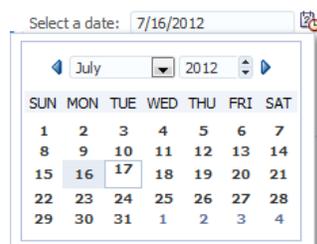
By default, the content delivery for the popup is lazy. When the user clicks the button, the `inputColor` component receives a PPR request, and rerenders, displaying a `chooseColor` component in a `popup` component.

Performance Tip:

If the `clientComponent` attribute on the `inputColor` component is set to `true`, then the popup and `chooseColor` component are delivered immediately. If the color palette is large, this could negatively affect initial page load performance.

The default color code format is the hexadecimal color format. However, you can override the format using a `ColorConverter` class.

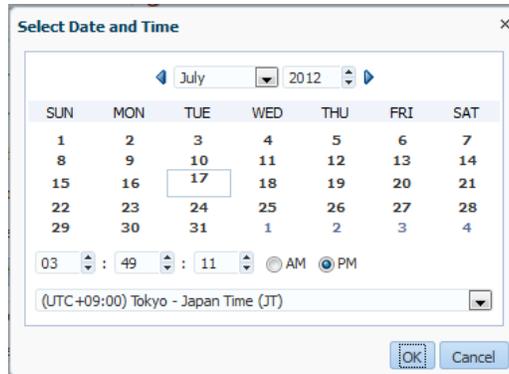
The `inputDate` component presents a text input field for entering dates and a button for picking dates from a popup calendar, as shown in [Figure 11-18](#). The default date format is the short date format appropriate for the current locale. For example, the default format in American English (ENU) is `mm/dd/yy`. However, you can override the format using a date-time converter (for information about using converters, see [Adding Conversion](#)).

Figure 11-18 inputDate Component

When you add a date-time converter and configure it to show both the date and the time, the date picker is displayed as a modal dialog with additional controls for the user

to enter a time. Additionally, if the converter is configured to show a time zone, a time zone dropdown list is shown in the dialog, as shown in [Figure 11-19](#).

Figure 11-19 Modal Dialog When Date-Time Converter Is Used



How to Add an inputColor Component

The `inputColor` component allows users either to enter a value in an input text field, or to select a color from a color chooser.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Color and Date Choosers](#).

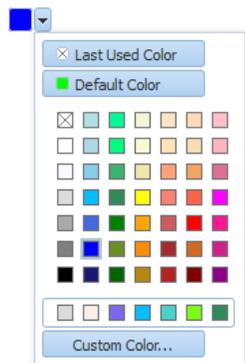
You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add an `inputColor` component:

1. In the Components window, from the Text and Selection panel, drag and drop an **Input Color** onto the page.
2. In Properties window, expand the Common section, and set the following:
 - **Label:** Specify a label for the component.
 - **Compact:** Set to `true` if you do not want to display the input text field, as shown in [Figure 11-20](#).

Figure 11-20 inputColor Component in Compact Mode

An inputColor in compact mode



3. Expand the **Data** section and set the following attributes:
 - **Value:** Specify the value of the component. If the EL binding for `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.
 - **ColorData:** Specify the list of colors to be displayed in the standard color palette. The number of provided colors can be 49 (7 colors x 7 colors), 64 (8 colors x 8 colors), or 121 (11 colors x 11 colors). The number set for this attribute will determine the valid value for the `width` attribute. For example, if you set the `colorData` attribute to 49, the width must be 7. If the number does not match the width, extra color elements in the list will be ignored and missing color elements will be displayed as no-color. The color list must be an array of type `TrColor` on the client side.
 - **CustomColorData:** Specify the list of custom-defined colors. The number of colors can be 7, 8, or 11. The color list must be an array of type `TrColor` on the client side. On the server side, it must be a `List` of `java.awt.Color` objects, or a list of hexadecimal color strings.
 - **DefaultColor:** Specify the default color using hexadecimal color code, for example `#000000`.
4. Expand the **Appearance** section and set the following attributes:
 - **Width:** Specify the width of the standard palette in cells. The valid values are 7, 8, and 11, which correspond to the values of the `colorData` and `customColorData` attributes.
 - **CustomVisible:** Specify whether or not the **Custom Color** button and custom color row are to be displayed. When set to `true`, the **Custom Color** button and custom color row will be rendered.
 - **DefaultVisible:** Specify whether or not the **Default** button is to be displayed. When set to `true`, the **Default** button will be rendered. The **Default** button allows the user to easily select the color set as the value for the `defaultColor` attribute.
 - **LastUsedVisible:** Specify whether or not the **Last Used** button is to be displayed. When set to `true` the **Last Used** button will be rendered, which allows the user to select the color that was most recently used.
 - **Editable:** Set to `onAccess` if you want the value of the component to appear as read-only until the user hovers over it. If you want the component to always

appear `editable`, select `always`. If you want the value to be inherited from an ancestor component, select `inherit`.

 **Note:**

If you select `inherit`, and no ancestor components define the `editable` value, then the value `always` is used.

- **Placeholder:** Specify the text that appears in the input component if the component is empty and does not have focus. When the component gets focus, or has a value, then the placeholder text is hidden.

The placeholder text is used to inform the user what should be entered in the input component.

 **Note:**

The placeholder value will only work on browsers that fully support HTML5.

5. If you want to style the label, expand the Style section and set **LabelStyle**. Enter a CSS style property and value for the label. For example, if you do not want the label text to wrap, you would enter `white-space: nowrap;` for the value.
6. Expand the Behavior section and set the following attribute:
 - **Chooseld:** Specify the ID of the `chooseColor` component which can be used to choose the color value. If not set, the `inputColor` component has its own default popup dialog with a `chooseColor` component.
 - **AutoComplete:** Set to `true` to allow the component to display previous values when the user begins to enter a matching value. Set to `false` if no matches should be displayed. Default is `true`.
 - **Usage:** Specify how the input component will be rendered in HTML 5 browser. The valid values are `auto`, `text`, and `search`. Default is `auto`.

If the usage type is `search`, the input component will render as an HTML 5 search input type. Some HTML 5 browsers may add a **Cancel** icon that can be used to clear the search text.

How to Add an InputDate Component

The `inputDate` component allows the user to either enter or select a date.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Color and Date Choosers](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add an `inputDate` component:

1. In the Components window, from the Text and Selection panel, drag and drop an **Input Date** onto the page.
2. In the Properties window, expand the Common section, and set the following:
 - **Label:** Specify a label for the component.
 - **Value:** Specify the value of the component. If the EL binding for `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.
3. In the Other section, set the following:
 - **Layout:** Specify the date picker functionality to expose on the `inputdate` component. This attribute provides fine control over the layout and ordering of various sections of the date picker. The sections that you can expose correspond to the **Layout** attribute values. You may combine the following values in any order to create a layout that makes sense for the given use case: `navigator`, `compactNavigator`, `date`, `time`, `compactTime`, `today`, `timeZone`.

For example, to create an `inputdate` component where the date picker layout specifies a mobile-friendly UI that displays a year and month navigation panel to select the date from a wall style calendar, and also allows time selection, enter the following attribute values without commas:

```
navigator date time
```

The attribute values that you can enter have the following purposes:

- `navigator`: Renders a mobile-friendly year and month navigation panel that utilizes the entire panel for tap navigation.
- `compactNavigator`: Renders a navigation panel in a single row with a dropdown and a number spinner for month and year selection. Note: This option reproduces the look and feel of past JDeveloper releases.
- `date`: Renders the wall calendar styled date selection grid.
- `time`: Renders a mobile-friendly time selection panel with spin wheels to select hours, minutes, seconds and meridian.
- `compactTime`: Renders a single row time selection panel with spinners to select hours, minutes, seconds and radio controls for meridian selection
- `today`: Renders a button labeled **Today** (or **Now** in time mode) as a quick shortcut to select the current date or time. This will submit the current date if enabled in a date only picker, else it will only switch to the current month in date+time selection mode.
- `timeZone`: Renders a select one choice dropdown with various time zones available for selection. Note: This option reproduces the look and feel of past JDeveloper releases and works well on mobile devices.

The skin property `-tr-layout` configures default values for the date picker layout if none is configured on the `inputdate` component.

- **Months:** Optionally, specify the number of months to be shown at once in the date picker. If you want to display more than one month, the additional months will be rendered side by side with the selected month appearing in the center. If no value is entered, the attribute value is based on the skin property `-tr-months`.

4. Optionally expand the Style section and set the **LabelStyle** attribute to a CSS style property and value. For example, if you did not want the label text to wrap, you would enter `white-space:nowrap;` as the value.
5. Expand the Data section and set the following attributes:
 - **MinValue:** Specify the minimum value allowed for the date value. When set to a fixed value on a tag, this value will be parsed as an ISO 8601 date. ISO 8601 dates are of the form "yyyy-MM-dd" (for example: 2002-02-15). All other uses require `java.util.Date` objects.
 - **MaxValue:** Specify the maximum value allowed for the date value. When set to a fixed value on a tag, this value will be parsed as an ISO 8601 date. ISO 8601 dates are of the form "yyyy-MM-dd" (for example: 2002-02-15). All other uses require `java.util.Date` objects.
 - **DisableDays:** Specify a binding to an implementation of the `org.apache.myfaces.trinidad.model.DateListProvider` interface. The `getDateList` method should generate a List of individual `java.util.Date` objects which will be rendered as disabled. The dates must be in the context of the given base calendar.

 **Performance Tip:**

This binding requires periodic roundtrips. If you just want to disable certain weekdays (for example, Saturday and Sunday), use the `disableDaysOfWeek` attribute.

- **DisableDaysOfWeek:** Specify a whitespace-delimited list of weekdays that should be rendered as disabled in every week. The list should consist of one or more of the following abbreviations: `sun, mon, tue, wed, thu, fri, sat`. By default, all days are enabled.
 - **DisableMonths:** Specify a whitespace-delimited list of months that should be rendered as disabled in every year. The list should consist of one or more of the following abbreviations: `jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec`. By default, all months are enabled.
 - **DefaultValue** : the default date would be used only when the picker is launched on an empty `DateTime` input field. When set to a literal value, this will be parsed as `yyyy-MM-dd hh:mm:ss` or `yyyy-MM-dd`. In case the time part is not specified, the time components (hours, minutes, seconds, milliseconds) will be zeroed-out. The created `Date` object will be in the application's time zone. See [What You May Need to Know About Including a Default Value for InputDate](#).
6. Expand the Behavior section and set the following:
 - **ChooseId.** Specify the ID of the `chooseDate` component which can be used to choose the date value. If not set, the `inputDate` component has its own default popup dialog with a `chooseDate` component.
 - **AutoComplete:** Set to `true` to allow the component to display previous values when the user begins to enter a matching value. Set to `false` if no matches should be displayed. Default is `true`.
 - **Usage:** Specify how the input component will be rendered in HTML 5 browser. The valid values are `auto, text, and search`. Default is `auto`.

If the usage type is `search`, the input component will render as an HTML 5 search input type. Some HTML 5 browsers may add a **Cancel** icon that can be used to clear the search text.

- Expand the Appearance section and set the following:
 - Editable:** Set to `onAccess` if you want the value of the component to appear as read-only until the user hovers over it. If you want the component to always appear editable, select `always`. If you want the value to be inherited from an ancestor component, select `inherit`.

 **Note:**

If you select `inherit`, and no ancestor components define the `editable` value, then the value `always` is used.

- Placeholder:** Specify the text that appears in the input component if the component is empty and does not have focus. When the component gets focus, or has a value, then the placeholder text is hidden.

The placeholder text is used to inform the user what should be entered in the input component.

 **Note:**

The placeholder value will only work on browsers that fully support HTML5.

- Optionally expand the Data section and set the **TimeZoneList** attribute to a **custom list of timezones**.

What You May Need to Know About Including a Default Value for an InputDate Component

You can configure the `inputDate` component to show a default value for either the date or both the date and the time by using the `DefaultValue` attribute. The value that you specify for this attribute is used to display the default values for date and time. You can specify one of the following values:

- Default date and time:** displays the user specified date and time as default date and time.
- Default date only:** displays the user specified date as default date.

 **Note:**

In case a default date and time are not specified, then today's date and current time will be shown as the default date and time.

You can specify the date and time either as an EL expression or as a literal value.

The following example shows how to specify a date and time default value for the `inputDate` component, by using an EL expression defined by the `testInput` backing bean.

```
<af:inputDate label="Date with default value" simple="true" id="inputDate"
    DefaultValue="#{testInput.defaultVal}">
    <af:convertDateTime type="both" dateStyle="full" pattern="yyyy-MM-dd
HH:mm:ss"/>
</af:inputDate>
```

The following example shows how to specify the default value for date and time as a literal value.

```
<af:inputDate label="Date with default value" simple="true" id="inputDate"
    DefaultValue="2019-01-01 05:05:05" >
    <af:convertDateTime type="both" dateStyle="full" pattern="yyyy-MM-dd
HH:mm:ss"/>
</af:inputDate>
```

The following example shows how to specify the default value for date only as a literal value.

```
<af:inputDate label="Date with default value" simple="true" id="inputDate"
    DefaultValue="2019-01-01" >
</af:inputDate>
```

What You May Need to Know About Setting the Time Value for an `inputDate` Component

When a user selects a date in the date picker, the page should automatically set the default time value within the same picker. You can use the `dateSelectionEventHandler` method to set the default time value for the `inputDate` component, when a user selects a date. To receive date-selection notifications, you must first register a client listener for the `dateSelection` event, as shown in the following sample.

```
<af:inputDate id="inputDate"
    <af:clientListener type="dateSelection" method="dateSelectionEventHandler"/>
    <af:convertDateTime type="both" dateStyle="short"/>
</af:inputDate>
```

What You May Need to Know About Selecting Time Zones Without the `inputDate` Component

By default, the `inputDate` component displays a drop down list of time zones if the associated converter is configured to do so, for example, if you include the time zone placeholder `z` in the converter's pattern. The user can only modify the time zone using this list. The list is configured to display the most common time zones.

However, there may be times when you need to display the list of time zones outside of the `inputDate` component. For example, on a `Application Preferences` page, you may want to use a `selectOneChoice` component that allows the user to select the time zone that will be used to display all `inputDates` in the application. A backing bean would handle the conversion between the time zone ID and the `java.util.TimeZone` object. Converters for the `inputDate` instances in the application would then bind the time zone to that time zone object.

You can access this list using either an API on the `DateTimeUtils` class, or using an EL expression on a component.

Following are the methods on `DateTimeUtils` class:

- `getCommonTimeZoneSelectItems ()`: Returns a list of commonly used time zones.
- `getCommonTimeZoneSelectItems (String timeZoneId)`: Returns a list of commonly used time zones, including the given time zone if it is not part of the list.

To access this list using EL, use one of the following expressions:

- `af:getCommonTimeZoneSelectItems`

For example:

```
<f:selectItems value="#{af:getCommonTimeZoneSelectItems()}" id="tzones2" />
```

- `af:getMergedTimeZoneSelectItems (id)`

For example:

```
<f:selectItems
value="#{af:getMergedTimeZoneSelectItems(demoInput.preferredTimeZoneId)}"
id="tzones" />
```

If you will be using an `inputDate` component and a selection list for its time zone on the same page, you must clear out the local value for the `inputDate`'s time zone to ensure that the value binding for the selection takes precedence. Otherwise, a non-null local value will take precedence, and the `inputDate` component will not appear to be updated. In the following example, the backing bean has a reference using the binding attribute to the `inputDate` component. When the user picks a new time zone, the ID is set and the code gets the converter for the `inputDate` and clears out its time zone. When the page is rendered, since the local value for the converter's time zone is null, it will evaluate `#{demoInput.preferredTimeZone}` and obtain the updated time zone.

```
<af:selectOneChoice label="Select a new timezone" id="soc1"
                    value="#{demoInput.preferredTimeZoneId}" autoSubmit="true">
  <f:selectItems
    value="#{af:getMergedTimeZoneSelectItems(demoInput.preferredTimeZoneId)}"
    id="tzones" />
</af:selectOneChoice>
<af:inputDate label="First inputDate with timezone bound" id="bound1"
              partialTriggers="tzpick" binding="#{demoInput.boundDate1}">
  <af:convertDateTime type="both" timeStyle="full"
                    timeZone="#{demoInput.preferredTimeZone}" />
</af:inputDate>
```

```
DemoInputBean.java
public void setPreferredTimeZoneId(String _preferredTimeZoneId)
{
    TimeZone tz = TimeZone.getTimeZone(_preferredTimeZoneId);
    setPreferredTimeZone (tz);
    this._preferredTimeZoneId = _preferredTimeZoneId;
}

public void setPreferredTimeZone(TimeZone _preferredTimeZone)
{
    this._preferredTimeZone = _preferredTimeZone;
    DateTimeConverter conv1 = (DateTimeConverter)
        _boundDate1.getConverter();
    conv1.setTimeZone(null);
}
```

What You May Need to Know About Multi-Selection Support in the chooseDate Component

With support from the `chooseDate` component load and selection events, you can define a client event listener to call an event handler and handle date selection based on the selection modifier keys used (Ctrl and Shift keys).

When the user first opens the `chooseDate` component or when the user navigates through the months of the calendar, the component generates `AdfChooseDateLoadEvent`. Then, when the user selects a date, the `chooseDate` component generates `AdfDateSelectionEvent` along with the selected `Date`. The generated selection event includes the selection modifier key information that your event handler can process.

The `AdfDateSelectionEvent` API provides the following methods to interact with the selection event:

- `getSelectedDate ()`: Gets the date in the date cell of the calendar that the user clicked when `AdfDateSelectionEvent` was generated.
- `getModifiers()`: Returns a list of the selection modifier keys with the following constant values:
 - `AdfRichChooseDate.SINGLE_SELECTION`: Default on desktops. Reflects user intention to select only 1 date. The previously selected date should get deselected and the current selected.
 - `AdfRichChooseDate.MULTI_SELECTION`: Default only on touch devices. Reflects the user intention to select an additional date. The previously selected date should remain selected and the current date added to the selections. If the current date is already selected, then `deselect/toggle`. This is triggered when the Control/Command modifier is pressed while the selection event is queued.
 - `AdfRichChooseDate.RANGE_SELECTION`: Reflects the user intention to select a range of dates. The previously selected date should remain selected and all the dates between the last selection to the present have to be selected. This is triggered when the Shift modifier is pressed while the selection event is queued.

Additionally, the API `setSelectedDates()` on the `RichChooseDate` client component along with component load and selection events allows you to implement multi-selection capability on the `chooseDate` component. The component will select the dates specified when you call `setSelectedDates()` on the generated event source. Load and selection events should be used to listen for user clicks (and key modifiers) to build your own collection of date selections and supply back to the component via the `setSelectedDates()` method.

For example, to register for date selection and load events, you can create a client listener for the `chooseDate` component:

```
<af:chooseDate id="chooseDate" >
  <af:clientListener type="dateSelection" method="dateSelectionEventHandler"/>
  <af:clientListener type="load" method="chooseDateLoadHandler"/>
  <af:convertDateTime type="both" dateStyle="short"/>
</af:chooseDate>
```

You can then define an event handler to handle the selected date based on the selection modifier keys. The following sample adds selected dates to the collection if the Ctrl key (MULTI_SELECTION) is pressed. If the Ctrl key is not pressed, then the previous collection is cleared and the selected date is added to it. And, if the Shift key (RANGE_SELECTION) is pressed, it selects all the dates in range.

```
<af:resource type="javascript">
  var dates = [];
  var minDate;
  var maxDate ;
  function dateSelectionEventHandler(event) {
    var eventSource = event.getSource();
    var selectedDate = event.getSelectedDate();
    var modifier = event.getModifiers();

    // ctrl or command is pressed
    if (modifier.indexOf(AdfRichChooseDate.MULTI_SELECTION ) != -1 ) {
      dates.push(selectedDate);
    }
    else if
      (modifier.indexOf(AdfRichChooseDate.RANGE_SELECTION) != -1) // shift is
pressed
      {
        if ( !minDate || ( minDate.getTime() > selectedDate.getTime())) {
          minDate = selectedDate;
        }
        if (!maxDate || ( maxDate.getTime() < selectedDate.getTime() )) // single
click {
          maxDate = selectedDate;
        }
        var timeDiff = Math.abs(maxDate.getTime() - minDate.getTime());
        var diffDays = Math.ceil(timeDiff / (1000 * 3600 * 24)) +1;
        dates = []
        for (var i = 0; i < diffDays; i++) {
          var selDate = new
            Date(minDate.getFullYear(),minDate.getMonth(), minDate.getDate() +
i, 0,0,0,0 );
          if (!eventSource.isDisabled(selDate)) {
            dates.push(selDate);
          }
        }
      }
    else if
      (modifier.indexOf(AdfRichChooseDate.SINGLE_SELECTION ) != -1 )
      {
        minDate = null;
        maxDate = null;
        dates = [];
        dates.push(selectedDate)
      }
    eventSource.setSelectedDates(dates);
  }

  function chooseDateLoadEventHandler(event) {
    var eventSource = event.getSource();
    eventSource.setSelectedDates(dates);
  }
</af:resource>
```

What You May Need to Know About Creating a Custom Time Zone List

The `inputDate` component can be configured to show a custom list of time zones using the **Time Zone List** attribute.

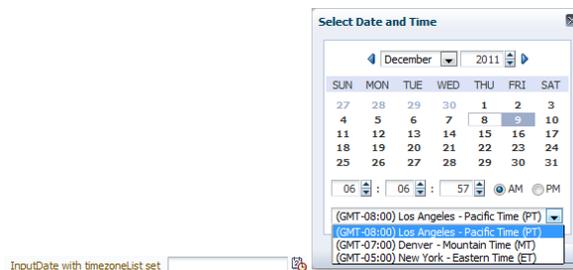
In the following example, the `inputDate` component uses the custom time zone list defined by the `testInput` backing bean to show only three US time zones.

```
<af:inputDate id="idtzl" label="InputDate with timezoneList set"
              timeZoneList="#{testInput.timezoneList}">
<af:convertDateTime type="both" timeStyle="full" timeZone="#{testInput.timeZone}/>
</af:inputDate>
```

```
testInput.java
public void setTimezoneList(List<String> _timezoneList)
{
    this._timezoneList = _timezoneList;
}
public List<String> getTimezoneList()
{
    return _timezoneList;
}
private List<String> _timezoneList = new ArrayList<String>
(Arrays.asList("America/Los_Angeles", "America/Denver", "America/New_York"));
```

Figure 11-21 illustrates the custom time zone list in the `inputDate` component defined by the backing bean.

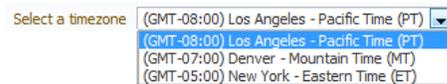
Figure 11-21 Custom Time Zone List in inputDate Component



You can also use the `getCustomTimeZoneSelectItems` helper EL method to get custom time zones as a list of `SelectItems`, which can then be used with components like `selectOneChoice`, as shown in the following example.

```
<af:selectOneChoice label="Select a timezone" id="soctzle1" value="America/New_York">
<f:selectItems value="#{af:getCustomTimeZoneSelectItems('America/New_York',
testInput.timezoneList)}" id="tzonesel" />
</af:selectOneChoice>
```

Figure 11-22 illustrates the custom time zone list in the `selectOneChoice` component defined by the `getCustomTimeZoneSelectItems` method.

Figure 11-22 Custom Time Zone List in selectOneChoice Component

The `getCustomTimeZoneSelectItems` helper EL method assumes that the input parameter list is sorted by timezone offset. For more information about `getCustomTimeZoneSelectItems` method, see the ADF Faces API documentation.

Using Selection Components

You can allow users to select single or multiple input values by adding ADF selection components in the form. Some of the different selection components are `selectBooleanCheckbox`, `selectBooleanRadio`, `selectManyCheckbox` and so on.

The selection components allow the user to select single and multiple values from a list or group of items. ADF Faces provides a number of different selection components, ranging from simple boolean radio buttons to list boxes that allow the user to select multiple items. The list of items within a selection component is made up of a number of `selectItem` components

All the selection components except the `selectItem` component delivers the `ValueChangeEvent` and `AttributeChangeEvent` events. The `selectItem` component only delivers the `AttributeChangeEvent` event. You must create a `valueChangeListener` handler or an `attributeChangeListener` handler, or both for them.

The `selectBooleanCheckbox` component value must always be set to a boolean and not an object. It toggles between selected and unselected states, as shown in [Figure 11-23](#).

Figure 11-23 selectBooleanCheckbox Component

The `selectBooleanCheckbox` component also provides a third mixed state that indicates that the component is neither selected or cleared, as shown in [Figure 11-24](#). Clicking a mixed state checkbox makes it selected. It toggles between the selected and cleared states, but it does not return to the mixed state by clicking, or any other click action.

Figure 11-24 selectBooleanCheckbox Component in Mixed State

The `selectBooleanRadio` component displays a boolean choice, and must always be set to a boolean. Unlike the `selectBooleanCheckbox` component, the `selectBooleanRadio` component allows you to group `selectBooleanRadio` components together using the same `group` attribute.

For example, say you have one boolean that determines whether or not a user is age 10 to 18 and another boolean that determines whether a user is age 19-100. As shown in [Figure 11-25](#), the two `selectBooleanRadio` components can be placed anywhere on the page, they do not have to be next to each other. As long as they share the same `group` value, they will have mutually exclusive selection, regardless of their physical placement on the page.



Tip:

Each `selectBooleanRadio` component must be bound to a unique boolean.

Figure 11-25 `selectBooleanRadio` Component

You use the `selectOneRadio` component to create a list of radio buttons from which the user can select a single value from a list, as shown in [Figure 11-26](#).

Figure 11-26 `selectOneRadio` Component

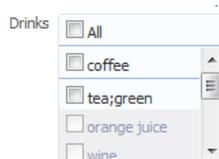
You use the `selectManyCheckbox` component to create a list of checkboxes from which the user can select one or more values, as shown in [Figure 11-27](#).

Figure 11-27 `selectManyCheckbox` Component

The `selectOneListbox` component creates a component which allows the user to select a single value from a list of items displayed in a shaded box, as shown in [Figure 11-28](#).

Figure 11-28 selectOneListbox Component

The `selectManyListbox` component creates a component which allows the user to select many values from a list of items. This component includes an **All** checkbox that is displayed at the beginning of the list of checkboxes, as shown in [Figure 11-29](#).

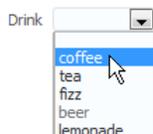
Figure 11-29 selectManyListbox Component

The `selectOneChoice` component creates a menu-style component, which allows the user to select a single value from a dropdown list of items. The `selectOneChoice` component is intended for a relatively small number of items in the dropdown list.

Best Practice:

If a large number of items is desired, use an `inputComboboxListOfValues` component instead. See [Using List-of-Values Components](#).

The `selectOneChoice` component is shown in [Figure 11-30](#).

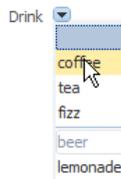
Figure 11-30 selectOneChoice Component

You can configure the `selectOneChoice` component to display in a compact mode, as shown in [Figure 11-31](#). When in compact mode, the input field is replaced with a smaller icon.

Figure 11-31 selectOneChoice Component in Compact Mode

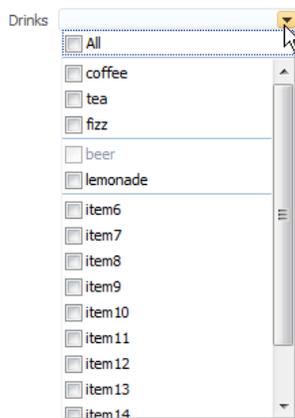
When the user clicks the icon, the dropdown list is displayed, as shown in [Figure 11-32](#).

Figure 11-32 List for selectOneChoice Component in Compact Mode



The `selectManyChoice` component creates a menu-style dropdown component, which allows the user to select multiple values from a dropdown list of items. This component can be configured to include an **All** selection item that is displayed at the beginning of the list of selection items. If the number of choices is greater than 15, a scrollbar will be presented, as shown in [Figure 11-33](#).

Figure 11-33 selectManyChoice Component



By default, all `selectItem` child components are built when the `selectManyChoice` component is built, as the page is rendered. However, if the way the list items are accessed is slow, then performance can be hampered. This delay can be especially troublesome when it is likely that the user will select the items once, and then not change them on subsequent visits.

For example, suppose you have a `selectManyChoice` component used to filter what a user sees on a page, and that the values for the child `selectItem` components are accessed from a web service. Suppose also that the user is not likely to change that selection each time they visit the page. By default, each time the page is rendered, all the `selectItems` must be built, regardless of whether or not the user will actually need to view them. Instead, you can change the `contentDelivery` attribute on the `selectManyChoice` component from `immediate` (the default) to `lazy`. The `lazy` setting causes the `selectItem` components to be built only when the user clicks the dropdown.

For both `immediate` and `lazy`, when the user then makes a selection, the values of the selected `selectItem` components are displayed in the field. However when `lazy`

content delivery is used, on subsequent visits, instead of pulling the selected values from the `selectItem` components (which would necessitate building these components), the values are pulled from the `lazySelectedLabel` attribute. This attribute is normally bound to a method that returns an array of `Strings` representing the selected items. The `selectItem` components will not be built until the user goes to view or change them, using the dropdown.

Note that there are limitations when using the lazy delivery method on the `selectManyChoice` component. For information about content delivery for the `selectManyChoice` component and its limitations, see [What You May Need to Know About the `contentDelivery` Attribute on the `SelectManyChoice` Component](#).

For the following components, if you want the label to appear above the control, you can place them in a `panelFormLayout` component.

- `selectOneChoice`
- `selectOneRadio`
- `selectOneListbox`
- `selectManyChoice`
- `selectManyCheckbox`
- `selectManyListbox`

For the following components, the attributes `disabled`, `immediate`, `readOnly`, `required`, `requireMessageDetail`, and `value` cannot be set from JavaScript on the client for security reasons (see [How to Set Property Values on the Client](#)):

- `selectOneChoice`
- `selectOneRadio`
- `selectOneListbox`
- `selectBooleanRadio`
- `selectBooleanCheckbox`
- `selectManyChoice`
- `selectManyCheckbox`
- `selectManyListbox`

How to Use Selection Components

The procedures for adding selection components are the same for each of the components. First, you add the selection component and configure its attributes. Then you add any number of `selectItem` components for the individual items in the list, and configure those.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Selection Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To use a selection component:

1. In the Components window, from the Text and Selection panel, drag and drop a selection component onto the page.
2. For all selection components except the `selectBooleanCheckbox` and `selectBooleanRadio` components, a dialog opens where you choose either to bind to a value in a managed bean, or to create a static list. On the second page of the dialog, you can set the following properties:
 - **Label:** Enter the label for the list.
 - **RequiredMessageDetail:** Enter the message that should be displayed if a selection is not made by the user. For information about messages, see [Displaying Hints and Error Messages for Validation and Conversion](#).
 - **Validator:** Enter an EL expression that resolves to a validation method on a managed bean (see [Validating and Converting Input](#)).
 - **Value:** Specify the value of the component. If the EL binding for the `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.

 **Note:**

If you are creating a `selectBooleanRadio` or `selectBooleanCheckbox` component, and you enter a value for the `value` attribute, you cannot also enter a value for the `selected` attribute, as it is a typesafe alias for the `value` attribute. You cannot use both.

- **ValueChangeListener:** Enter an EL expression that resolves to a listener on a managed bean that handles value change events.
3. Expand the Appearance section of the Properties window and set the attributes, as described in [Table 11-1](#). Note that only attributes specific to the selection components are discussed here. Many of the attributes are the same as for input text components. See [How to Add an inputText Component](#).

Table 11-1 Appearance Attributes for Selection Components

Components	Attribute
<code>selectOneRadio</code> , <code>selectManyCheckbox</code>	Layout: Set to <code>vertical</code> to have the buttons or checkboxes displayed vertically. Set to <code>horizontal</code> to have them displayed in a single horizontal line.
<code>selectManyListbox</code>	Size: Set to the number of items that should be displayed in the list. If the number of items in the list is larger than the <code>size</code> attribute value, a scrollbar will be displayed.
<code>selectManyListbox</code> , <code>selectManyChoice</code>	SelectAllVisible: Set to <code>true</code> to display an All selection that allows the user to select all items in the list.
<code>selectOneChoice</code>	Mode: Set to <code>compact</code> to display the component only when the user clicks the dropdown icon.

Table 11-1 (Cont.) Appearance Attributes for Selection Components

Components	Attribute
selectOneRadio, selectOneListbox, selectOneChoice	UnselectedLabel: Enter text for the option that represents a value of null, meaning nothing is selected. If <code>unselectedLabel</code> is not set and if the component does not have a selected value, then an option with an empty string as the label and value is rendered as the first option in the choice box (if there is not an empty option already defined). Note that you should set the <code>required</code> attribute to <code>true</code> when defining an <code>unselectedLabel</code> value. If you do not, two blank options will appear in the list. Once an option has been successfully selected, and if <code>unselectedLabel</code> is not set, then the empty option will not be rendered.

- Expand the Behavior section of the Properties window and set the attributes, as described in [Table 11-2](#). Note that only attributes specific to the selection components are discussed here. Many of the attributes are the same as for input text components. See [How to Add an inputText Component](#).

Table 11-2 Behavior Attributes for Selection Components

Component	Attribute
All except the boolean selection components	ValuePassThru: Specify whether or not the values are passed through to the client. When <code>valuePassThru</code> is <code>false</code> , the value and the options' values are converted to indexes before being sent to the client. Therefore, when <code>valuePassThru</code> is <code>false</code> , there is no need to write your own converter when you are using custom Objects as your values, options, or both. If you need to know the actual values on the client-side, then you can set <code>valuePassThru</code> to <code>true</code> . This will pass the values through to the client, using your custom converter if it is available; a custom converter is needed if you are using custom objects. The default is <code>false</code> . Note that if your selection components uses ADF Model binding, this value will be ignored.
selectBooleanRadio	Group: Enter a group name that will enforce mutual exclusivity for all other <code>selectBooleanRadio</code> components with the same <code>group</code> value.

- If you want the value of a `selectOneChoice` or `selectManyChoice` component to appear as read-only until the user hovers over it, expand the Appearance section and set **Editable** to `onAccess`. If you want the component to always appear editable, select `always`. If you want the value to be inherited from an ancestor component, select `inherit`.

 **Note:**

If you select `inherit`, and no ancestor components define the `editable` value, then the value `always` is used.

6. If you do not want the child `selectItem` components for the `selectManyChoice` to be built each time the page is rendered, do the following:
 - Create logic that can store the labels of the selected items and also return those labels as an array of strings.
 - Expand the Advanced section, and set **ContentDelivery** to lazy.
 - Bind **LazySelectedLabel** to the method that returns the array of the selected items.

Note that there are limitations to using lazy content delivery. For information about content delivery for the `selectManyChoice` component, see [What You May Need to Know About the contentDelivery Attribute on the SelectManyChoice Component](#).

7. If you want the `af:selectBooleanCheckbox` component to show the indeterminate (or mixed) state that indicates that the component is neither selected or cleared, expand the Advanced section, and set the **nullValueMeans** attribute to `mixed`.

The user cannot make the `selectBooleanCheckbox` component into the mixed state with a single click. For example, a checkbox can be used to show the mixed state when some, not all, children options of the checkbox are enabled or disabled. The mixed state changes to the selected state when all its children options are enabled, and it changes to the unselected state when all the children options under it are disabled. This behavior is not automatic and needs to be managed by backend application code.
8. For the boolean components, drag and drop any number of `selectItem` components as children to the boolean component. These will represent the items in the list (for other selection components, the dialog in Step 2 automatically added these for you).
9. With the `selectItem` component selected, in the Properties window, expand the **Common** section, and if not set, enter a value for the `value` attribute. This will be the value that will be submitted.
10. Expand the Appearance section, and if not set, enter a value for **Label**. This will be the text that is displayed in the list.
11. Expand the Behavior section, and set **Disabled** to `true` if you want the item to appear disabled in the list.

What You May Need to Know About the contentDelivery Attribute on the SelectManyChoice Component

When the `contentDelivery` attribute on the `selectManyChoice` component is set to `immediate` (the default), the following happens:

- First visit to the page:
 - The `selectManyChoice` and all `selectItem` components are built as the page is rendered. This can cause performance issues if there are many items, or if the values for the `selectItem` components are accessed for example, from a web service.
 - When the `selectManyChoice` component renders, nothing displays in the field, as there has not yet been a selection.
 - When user clicks drop down, all items are shown.

- When user selects items, the corresponding labels for the selected `selectItem` components are shown in field.
- When page is submitted, values are posted back to the model.
- Subsequent visit: The `selectManyChoice` and all `selectItem` components are built again as the page is rendered. Labels for selected `selectItem` components are displayed in field. This will cause the same performance issues as on the first visit to the page.

When the `contentDelivery` attribute on the `selectManyChoice` component is set to `lazy`, the following happens:

- First visit to the page:
 - The `selectManyChoice` is built as the page is rendered, but the `selectItem` components are not.
 - When the `selectManyChoice` component renders, nothing displays in the field, as there has not yet been a selection.
 - When user clicks drop down, the `selectItem` components are built. While this is happening, the user sees a "busy" spinner. Once the components are built, all items are shown.
 - When user selects items, the corresponding labels for the selected `selectItem` components are shown in field.
 - When page is submitted, values are posted back to the model.
- Subsequent visit:
 - When page is first rendered, only the `selectManyChoice` component is built. At this point, the value of the `lazySelectedLabel` attribute is used to display the selected items.
 - If user clicks drop down, the `selectItem` components are built. While this is happening, the user sees a "busy" spinner. Once the components are built, all items are shown.

Once the `selectItem` components are built, the `selectManyChoice` component will act as though its `contentDelivery` attribute is set to `immediate`, and use the actual value of the `selectItem` components to display the selected items.

Following are limitations for using lazy content delivery for the `selectManyChoice` component:

- You cannot store the value of the `selectManyChoice` in Request scope. On postback, the value attribute is accessed from the model, rather than decoding what was returned from the client. If the value is stored in Request scope, that value will be empty. Do not store the value in Request scope.
- On postbacks, converters are not called. If you are relying on converters for postbacks, then you should not use lazy content delivery.

Using Shuttle Components

The ADF shuttle components comprises of two list boxes and buttons to move items from one list box to the other. You can allow users to make a single or multiple selection from the available list of values box and add it to the selected list of values box.

The `selectManyShuttle` and `selectOrderShuttle` components present the user with two list boxes and buttons to move or shuttle items from one list box to the other. The user can select a single item or multiple items to shuttle between the leading (**Available values**) list box and the trailing (**Selected values**) list box. For either component, if you want the label to appear above the control, place them in a `panelFormLayout` component.

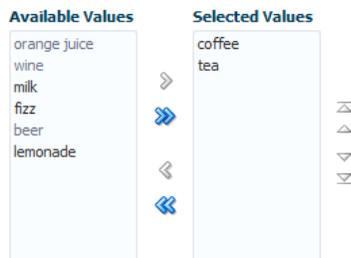
The `selectManyShuttle` component is shown in [Figure 11-34](#).

Figure 11-34 `selectManyShuttle` component



The `selectOrderShuttle` component additionally includes up and down arrow buttons that the user can use to reorder values in the **Selected values** list box, as shown in [Figure 11-35](#). When the list is reordered, a `ValueChangeEvent` event is delivered. If you set the `readOnly` attribute to `true`, ensure the values to be reordered are selected values that will be displayed in the trailing list (**Selected values**).

Figure 11-35 `selectOrderShuttle` Component



The `value` attribute of these components, like any other `selectMany` component, must be a `List` or an `Array` of values that correspond to a value of one of the contained `selectItem` components. If a value of one of the `selectItems` is in the `List` or `Array`, that item will appear in the trailing list. You can convert a `selectManyListbox` component directly into a `selectManyShuttle`; instead of the `value` driving which items are selected in the listbox, it affects which items appear in the trailing list of the `selectOrderShuttle` component.

Similar to other `select` components, the `List` or `Array` of items are composed of `selectItem` components nested within the `selectManyShuttle` or `selectOrderShuttle` component. The following example shows a sample `selectOrderShuttle` component that allows the user to select the top five file types from a list of file types.

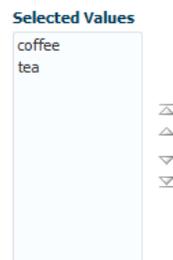
```

<af:selectOrderShuttle value="#{helpBean.topFive}"
  leadingHeader="#{explorerBundle['help.availableFileTypes']}"
  trailingHeader="#{explorerBundle['help.top5']}"
  simple="true" id="sos1">
  <af:selectItem label="XLS" id="si1"/>
  <af:selectItem label="DOC" id="si2"/>
  <af:selectItem label="PPT" id="si3"/>
  <af:selectItem label="PDF" id="si4"/>
  <af:selectItem label="Java" id="si5"/>
  <af:selectItem label="JWS" id="si6"/>
  <af:selectItem label="TXT" id="si7"/>
  <af:selectItem label="HTML" id="si8"/>
  <af:selectItem label="XML" id="si9"/>
  <af:selectItem label="JS" id="si10"/>
  <af:selectItem label="PNG" id="si11"/>
  <af:selectItem label="BMP" id="si12"/>
  <af:selectItem label="GIF" id="si13"/>
  <af:selectItem label="CSS" id="si14"/>
  <af:selectItem label="JPR" id="si15"/>
  <af:selectItem label="JSPX" id="si16"/>
  <f:validator validatorId="shuttle-validator"/>
</af:selectOrderShuttle>

```

If you set the `reorderOnly` attribute of a `selectOrdershuttle` component to `true`, the shuttle function will be disabled, and only the **Selected Values** listbox appears. The user can only reorder the items in the listbox, as shown in [Figure 11-36](#).

Figure 11-36 `selectOrderShuttle` Component in Reorder-Only Mode



How to Add a `selectManyShuttle` or `selectOrderShuttle` Component

The procedures for adding shuttle components are the same for both components. First you add the selection component and configure its attributes. Then you add any number of `selectItem` components for the individual items in the list, and configure those.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Shuttle Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add a `selectManyShuttle` or `selectOrderShuttle` component:

1. In the Components window, from the Text and Selection panel, drag and drop a **Shuttle** or **Shuttle (Ordered)** onto the page.

2. A dialog appears where you choose either to bind to a value in a managed bean, or to create a static list. On the second page of the dialog, you can set the following:
 - **Label:** Enter the label for the list.
 - **RequiredMessageDetail:** Enter the message that should be displayed if a selection is not made by the user. For information about messages, see [Displaying Hints and Error Messages for Validation and Conversion](#).
 - **Size:** Specify the display size (number of items) of the lists. The size specified must be between 10 and 20 items. If the attribute is not set or has a value less than 10, the size would have a default or minimum value of 10. If the attribute value specified is more than 20 items, the size would have the maximum value of 20.
 - **Validator:** Enter an EL expression that resolves to a validation method on a managed bean.
 - **Value:** Specify the value of the component. If the EL binding for the `value` points to a bean property with a `get` method but no `set` method, the component will be rendered in read-only mode.
 - **ValueChangeListener:** Enter an EL expression that resolves to a listener on a managed bean that handles value change events.
3. In the Properties window, expand the Appearance section and set the following:
 - **Layout:** Specify whether the component will be in horizontal or vertical layout. The default is `horizontal`, meaning the leading and trailing list boxes are displayed next to each other. When set to `vertical`, the leading list box is displayed above the trailing list box.
 - **LeadingHeader:** Specify the header text of the leading list of the shuttle component.
 - **LeadingDescShown:** Set to `true` to display a description of the selected item at the bottom of the leading list box.
 - **TrailingHeader:** Specify the header of the trailing list of the shuttle component.
 - **TrailingDescShown:** Set to `true` to display a description of the selected item at the bottom of the trailing list box.
4. Expand the Behavior section and optionally set the following attributes:
 - **ValuePassThru:** Specify whether or not the values are passed through to the client. When `valuePassThru` is `false`, the value and the options' values are converted to indexes before being sent to the client. Therefore, when `valuePassThru` is `false`, there is no need to write your own converter when you are using custom objects as your values, options, or both. If you need to know the actual values on the client-side, then you can set `valuePassThru` to `true`. This will pass the values through to the client, using your custom converter if it is available; a custom converter is needed if you are using custom objects. The default is `false`.
 - **ReorderOnly** (`selectOrderShuttle` component only): Specify whether or not the shuttle component is in reorder-only mode, where the user can reorder the list of values, but cannot add or remove them.
5. In the Structure window, select one of the `selectItem` components, and in the Properties window, set any needed attributes.

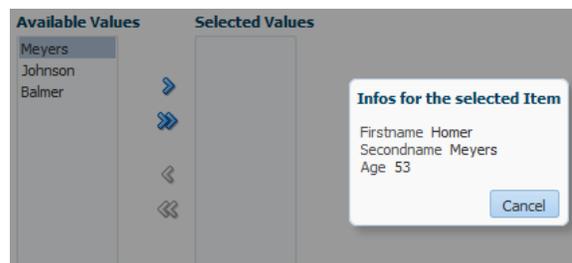
**Tip:**

If you elected to have the leading or trailing list box display a description, you must set a value for the `shortDesc` attribute for each `selectItem` component.

What You May Need to Know About Using a Client Listener for Selection Events

You can provide the user with information about each selected item before the user shuttles it from one list to another list by creating JavaScript code to perform processing in response to the event of selecting an item. For example, your code can obtain additional information about that item, then display it as a popup to help the user make the choice of whether to shuttle the item or not. [Figure 11-37](#) shows a `selectManyShuttle` component in which the user selects **Meyers** and a popup provides additional information about this selection.

Figure 11-37 `selectManyShuttle` with `selectionListener`



You implement this feature by adding a client listener to the `selectManyShuttle` or `selectOrderShuttle` component and then create a JavaScript method to process this event. The JavaScript code is executed when a user selects an item from the lists. For information about using client listeners for events, see [Listening for Client Events](#).

How to add a client listener to a shuttle component to handle a selection event:

1. In the Components window, from the Operations panel, in the Listeners group, drag a **Client Listener** and drop it as a child to the shuttle component.
2. In the Insert Client Listener dialog, enter a function name in the **Method** field (you will implement this function in the next step), and select `propertyChange` from the **Type** dropdown.

If for example, you entered **showDetails** as the function, JDeveloper would enter the code shown in bold in the following example.

```
<af:selectManyShuttle value="#{demoInput.manyListValue1}" id="sms1"
  valuePassThru="true" ...>
  <af:clientListener type="propertyChange" method="showDetails"/>
  <af:selectItem label="coffee" value="bean" id="sil" />
  ...
</af:selectManyShuttle>
```

This code causes the `showDetails` function to be called any time the property value changes.

3. In your JavaScript, implement the function entered in the last step. This function should do the following:
 - Get the shuttle component by getting the source of the event.
 - Use the client JavaScript API calls to get information about the selected items.

In the following example, `AdfShuttleUtils.getLastSelectionChange` is called to get the value of the last selected item

```
function showDetails(event)
{
  if(AdfRichSelectManyShuttle.SELECTION == event.getPropertyName())
  {
    var shuttleComponent = event.getSource();
    var lastChangedValue =
AdfShuttleUtils.getLastSelectionChange(shuttleComponent,
event.getOldValue());
    var side = AdfShuttleUtils.getSide(shuttleComponent, lastChangedValue);
    if(AdfShuttleUtils.isSelected(shuttleComponent, lastChangedValue))
    {
      //do something...
    }
    else
    {
      //do something else
    }
    if(AdfShuttleUtils.isLeading(shuttleComponent, lastChangedValue))
    {
      //queue a custom event (see serverListener) to call a java method on the server
    }
  }
}
```

Using the richTextEditor Component

Using the ADF `richTextEditor` component you can allow users to edit and add formatted input content. The `richTextEditor` provides several options to format the content.

The `richTextEditor` component provides an input field that can accept text with formatting. It also supports label text, and messages. It allows the user to change font name, size, and style, create ordered lists, justify text, and use a variety of other features. The `richTextEditor` component also can be used to edit an HTML source file. Two command buttons are used to toggle back and forth between editing standard formatted text and editing the HTML source file. [Figure 11-38](#) shows the rich text editor component in standard rich text editing Mode.

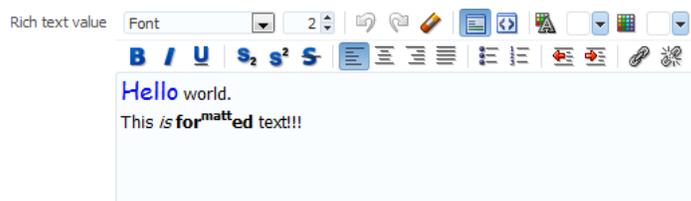
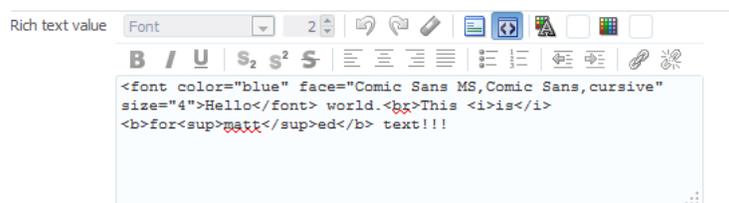
Figure 11-38 The richTextEditor Component in Standard Editing Mode

Figure 11-39 shows the editor in source code editing mode.

Figure 11-39 richTextEditor in Source Editing Mode

Other supported features include:

- Font type
- Font size
- Link/unlink
- Clear styling
- Undo/redo
- Bold/italics/underline
- Subscript/superscript
- Justify (left, middle, right, full)
- Ordered/unordered lists
- Indentation
- Text color/background color
- Rich text editing mode / source code editing mode
- Rich text editing toolbar inline display mode / popup display mode

The value (entered text) of the rich text editor is a well-formed XHTML fragment. Parts of the value may be altered for browser-specific requirements to allow the value to be formatted. Also, for security reasons, some features such as script-related tags and attributes will be removed. There are no guarantees that this component records only the minimal changes made by the user. Because the editor is editing an XHTML document, the following elements may be changed:

- Nonmeaningful whitespace
- Element minimization

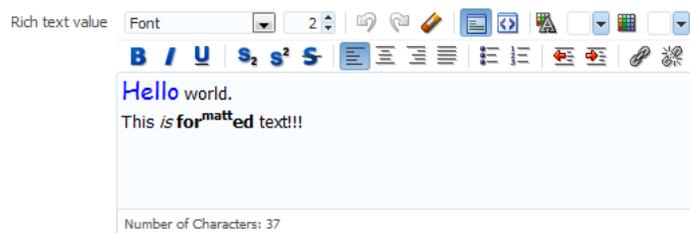
- Element types
- Order of attributes
- Use of character entities

The editor supports only HTML 4 tags, with the exception of:

- Script, noscript
- Frame, frameset, noframes
- Form-related elements (input, select, optgroup, option, textarea, form, button, label, isindex)
- Document-related elements (html, head, body, meta, title, base, link)

The `richTextEditor` component provides a footer facet that you can use to display additional information, or to add user interface elements. For example, [Figure 11-40](#) shows the `richTextEditor` component with a character counter in its footer facet.

Figure 11-40 The richTextEditor Component with Character Counter in Footer Facet



The `richTextEditor` component also supports tags that pull in content (such as `applet`, `iframe`, `object`, `img`, and `a`). For the `iframe` tag, the content should not be able to interact with the rest of the page because browsers allow interactions only with content from the same domain. However, this portion of the page is not under the control of the application.

While the `richTextEditor` component does not support font units such as `px` and `em`, it does support font size from 1 to 7 as described in the HTML specification. It does not support `embed` or unknown tags (such as `<foo>`).

On the client, the `richTextEditor` component does not support `getValue` and `setValue` methods. There is no guarantee the component's value on the client is the same as the value on the server. Therefore, the `richTextEditor` does not support client-side converters and validators. Server-side converters and validators will still work.

The rich text editor delivers `ValueChangeEvent` and `AttributeChangeEvent` events. Create `valueChangeListener` and `attributeChangeListener` handlers for these events as required.

You can also configure the `richTextEditorInsertBehavior` tag, which works with command components to insert given text into the `richTextEditor` component. The text to be entered can be a simple string, or it can be preformatted text held, for example, in a managed bean.

By default, the toolbar in the `richTextEditor` component allows the user to change many aspects of the text, such as the font, font size and weight, text alignment, and view mode, as shown in [Figure 11-41](#).

Figure 11-41 Toolbar in `richTextEditor` Component



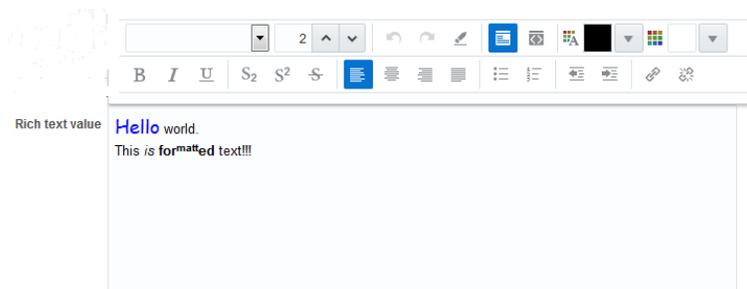
[Figure 11-42](#) shows a toolbar that has been customized. Many of the toolbar buttons have been removed and a toolbar with a custom toolbar button and a menu have been added.

Figure 11-42 Customized Toolbar for `richTextEditor`



As [Figure 11-43](#) shows, you can configure the toolbar to display in popup mode when displaying the toolbar inline (default behavior) with the `richTextEditor` is not desired.

Figure 11-43 Toolbar Display Mode Set to Popup



How to Add a richTextEditor Component

Once you add a `richTextEditor` component, you can configure it so that text can be inserted at a specific place, and you can also customize the toolbar. See [How to Add the Ability to Insert Text into a richTextEditor Component](#), and [How to Customize the Toolbar](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using the richTextEditor Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add a `richTextEditor` component:

1. In the Components window, from the Text and Selection panel, drag and drop a **Rich Text Editor** onto the page.
2. In the Properties window, expand the Common section, and set the `value` attribute.
3. Expand the Appearance section and set the following:
 - **Rows:** Specify the height of the edit window as an approximate number of characters shown.
 - **Columns:** Specify the width of the edit window as an approximate number of characters shown.
 - **Label:** Specify a label for the component.
4. Expand the Behavior section and set the following:
 - **EditMode:** Select whether you want the editor to be displayed using the WYSIWYG or source mode.
 - **ContentDelivery:** Specify whether or not the data within the editor should be fetched when the component is rendered initially. When the `contentDelivery` attribute value is `immediate`, data is fetched and displayed in the component when it is rendered. If the value is set to `lazy`, data will be fetched and delivered to the client during a subsequent request. See [Content Delivery](#).
5. Expand the Other section and set the **DimensionsFrom** property to one of the following:

- `auto`: Resize the `richTextEditor` component and get dimensions from its container component, or its own content.

If the parent component to `richTextEditor` component allows stretching of its child, then the `richTextEditor` component will stretch to fill the parent component. If the parent component does not allow stretching, then the `richTextEditor` component gets its dimensions from the content.

If the `richTextEditor` component is inside a container component, it gets its dimensions from its parent component. If there is no parent component, the `richTextEditor` component gets resized as per its content.

- `content`: Resize the `richTextEditor` component and get dimensions from its own content. This is the default value.
- `parent`: Resize the `richTextEditor` component and get its dimensions from `inlineStyle`.

If `inlineStyle` is not specified, it automatically gets its dimensions from the container component. If there is no container component and `inlineStyle` is not specified, the `richTextEditor` component gets its dimensions from the specified skin.

For information about how components stretch, see [Geometry Management and Component Stretching](#).

 **Note:**

You can also specify an EL expression that evaluates to a String value of `auto`, `content`, or `parent`.

 **Tip:**

You can set the width of a `richTextEditor` component to full width or 100%. However, this works reliably only if the editor is contained in a geometry-managing parent components. It may not work reliably if it is placed in flowing layout containers such as `panelFormLayout` or `panelGroupLayout`. See [Geometry Management and Component Stretching](#).

How to Add the Ability to Insert Text into a richTextEditor Component

To allow text to be inserted into a `richTextEditor` component, add the `richTextEditorInsertBehavior` tag as a child to a command component that will be used to insert the text.

Before you begin:

It may be helpful to have an understanding of the rich text editor component. See [Using the richTextEditor Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

You need to create a `richTextEditor` component as described in [How to Add a richTextEditor Component](#). Set the `clientComponent` attribute to `true`.

To add text insert behavior:

1. Add a command component that the user will click to insert the text. For procedures, see [How to Use Buttons and Links for Navigation and Deliver ActionEvents](#).
2. In the Components window, from the Operations panel, in the Behavior group, drag and drop a **Rich Text Editor Insert Behavior** as a child to the command component.
3. In the Rich Text Editor Insert Behavior dialog, enter the following:
 - **For:** Use the dropdown arrow to select **Edit** and then navigate to select the `richTextEditor` component into which the text will be inserted.
 - **Value:** Enter the value for the text to be inserted. If you want to insert static text, then enter that text. If you want the user to be able to insert the value of another component (for example, the value of a `selectOneChoice` component), then enter an EL expression that resolves to that value. If you want the user to enter preformatted text, enter an EL expression that resolves to that text. For example the following code shows preformatted text as the value for an attribute in the `demoInput` managed bean.

```
private static final String _RICH_INSERT_VALUE =
    "<p align=\"center\" style=\"border: 1px solid gray;
      margin: 5px; padding: 5px;\">" +
    "<font size=\"4\"><span style=\"font-family: Comic Sans MS,
      Comic Sans,cursive;\">Store Hours</span></font><br/>\n" +
    "<font size=\"1\">Monday through Friday 'til 8:00 pm</font><br/>\n" +
    "<font size=\"1\">Saturday & Sunday 'til 5:00 pm</font>" +
    "</p>";
```

The following example shows how the text is referenced from the `richTextEditorInsertBehavior` tag.

```
<af:richTextEditor id="idRichTextEditor" label="Rich text value"
    value="#{demoInput.richValue2}"/>
. . .
</af:richTextEditor>
<af:button text="Insert Template Text" id="b1">
    <af:richTextEditorInsertBehavior for="idRichTextEditor"
        value="#{demoInput.richInsertValue}"/>
</af:button>
```

4. By default, the text will be inserted when the action event is triggered by clicking the command component. However, you can change this to another client event by choosing that event from the dropdown menu for the `triggerType` attribute.

How to Customize the Toolbar

Place the toolbar and toolbar buttons you want to add in custom facets that you create. Then, reference the facet (or facets) from an attribute on the toolbar, along with keywords that determine how or where the contained items should be displayed.

To allow text to be inserted into a `richTextEditor` component, add the `richTextEditorInsertBehavior` tag as a child to a command component that will be used to insert the text.

Before you begin:

It may be helpful to have an understanding of the rich text editor component. See [Using the richTextEditor Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To customize the toolbar:

1. In the JSF page of the Components window, from the Core panel, drag and drop a **Facet** for each section of the toolbar you want to add.

For example, to add the custom buttons shown in [Figure 11-42](#), you would add two `<f:facet>` tags. Ensure that each facet has a unique name for the page.

Tip:

To ensure that there will be no conflicts with future releases of ADF Faces, start all your facet names with `customToolbar`.

2. In the ADF Faces page of the Components window, from the Menus and Toolbars panel, drag and drop a **Toolbar** into each facet and add toolbar buttons or other components and configure as needed.

For information about toolbars and toolbar buttons, see [Using Toolbars](#).

3. With the `richTextEditor` component selected, in the Properties window, in the Appearance section, click the dropdown icon for the `toolboxLayout` attribute and choose **Edit**.
4. In the Edit Property: `ToolboxLayout` dialog, add facet names in the order in which you want the contents in the custom facets to appear. In addition to those facets,

you can also include all, or portions, of the default toolbar, using the following keywords:

- `all`: All the toolbar buttons and text in the default toolbar. If `all` is entered, then any keyword for noncustom buttons will be ignored.
- `font`: The font selection and font size buttons.
- `history`: Undo and redo buttons.
- `mode`: Rich text mode and source code mode buttons.
- `color`: Foreground and background color buttons.
- `formatAll`: Bold, italic, underline, superscript, subscript, strikethrough buttons. If `formatAll` is specified, `formatCommon` and `formatUncommon` will be ignored.
- `formatCommon`: Bold, italic, and underline buttons.
- `formatUncommon`: Superscript, subscript, and strikethrough buttons.
- `justify`: Left, center, right and full justify buttons.
- `list`: Bullet and numbered list buttons.
- `indent`: Outdent and indent buttons.
- `link`: Add and remove Link buttons.

For example, if you created two facets named `customToolbar1` and `customToolbar2`, and you wanted the complete default toolbar to appear in between your custom toolbars, you would enter the following list:

- `customToolbar1`
- `all`
- `customToolbar2`

You can also determine the layout of the toolbars using the following keywords:

- `newline`: Places the toolbar in the next named facet (or the next keyword from the list in the `toolbarLayout` attribute) on a new line. For example, if you wanted the toolbar in the `customToolbar2` facet to appear on a new line, you would enter the following list:

- `customToolbar1`
- `all`
- `newline`
- `customToolbar2`

If instead, you did not want to use all of the default toolbar, but only the font, color, and common formatting buttons, and you wanted those buttons to appear on a new line, you would enter the following list:

- `customToolbar1`
- `customToolbar2`
- `newline`
- `font`
- `color`
- `formatCommon`

- `stretch`: Adds a spacer component that stretches to fill all available space so that the next named facet (or next keyword from the default toolbar) is displayed as right-aligned in the toolbar.

About richTextEditor for UIWebView User-Agent

`UIWebView` is a user interface control used in iOS applications. It allows a developer to embed a web browser within the ADF application, allowing access to web pages without leaving the application.

ADF Faces now supports the default `UIWebView` user-agent for `richTextEditor` in ADF Faces applications. The `UIWebView` user-agent is a short string that applications send to identify themselves to web servers when accessing different HTML pages.

When your ADF application connects to a website, it includes a `User-Agent` field in its HTTP header. The contents of the user-agent field vary from application to application. Each application has its own, distinctive user-agent. Essentially, a user-agent identifies the browser version and the operating system of the device.

The web server can use this information to serve different web pages to different web browsers and different operating systems. For example, a website could send mobile pages to mobile browsers or send a `Please upgrade your browser` message to supported browsers. The following example shows a typical `UIWebView` user-agent for an application:

```
Mozilla/5.0 (iPhone; CPU iPhone OS 8_2 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Mobile/12D508
```

Note:

Oracle ADF supports different browsers for desktop and mobile devices. For the list of supported browsers, click the Certification Information link that appears for your release on the OTN Documentation page at <http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html>.

Using File Upload

The ADF allows you to provide the input by uploading single or multiple files. You can also update the already submitted file.

The `inputFile` component provides users with file uploading and updating capabilities. This component allows the user to select a local file and upload it to a selectable location on the server (to download a file from the server to the user, see [How to Use an Action Component to Download Files](#)).

The `inputFile` component delivers the standard `ValueChangeEvent` event as files are being uploaded, and it manages the loading process transparently. The `value` property of an `inputFile` component is set to an instance of the `org.apache.myfaces.trinidad.model.UploadedFile` class when the file is uploaded.

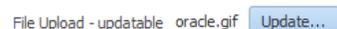
To initiate the upload process you first must configure the page's form to allow uploads. You then create an action component such as a command button, as shown in [Figure 11-44](#), that can be used to upload a file.

Figure 11-44 inputFile Component



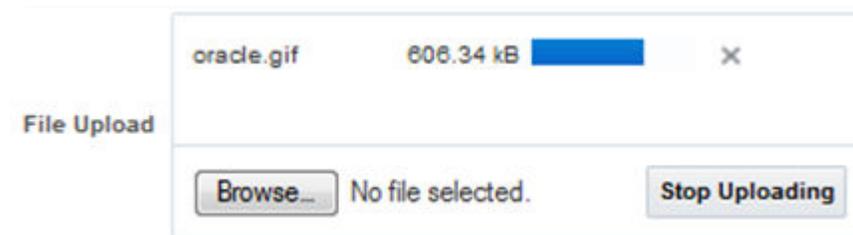
Once a file has been uploaded, and so the value of the `inputFile` is not null (either after the initial load is successful or it has been specified as an initial value), you can create an **Update** button that will be displayed instead of the **Browse** button, as shown in [Figure 11-45](#). This will allow the user to modify the value of the `inputFile` component.

Figure 11-45 inputFile Component in Update Mode



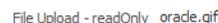
Setting the `uploadType` attribute to **auto** or **manual** displays the progress bar while uploading a file as shown in [Figure 11-46](#).

Figure 11-46 inputFile Component with Progress Bar

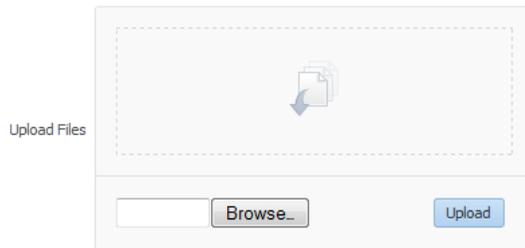


You can also specify that the component be able to load only a specific file by setting the `readOnly` property to `true`. In this mode, only the specified file can be loaded, as shown in [Figure 11-47](#).

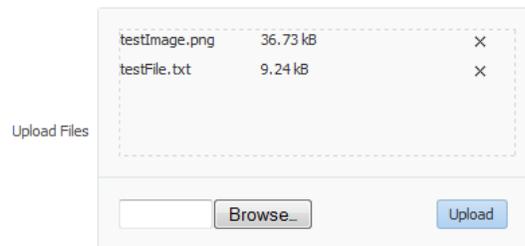
Figure 11-47 inputFile Component in Read-Only Mode



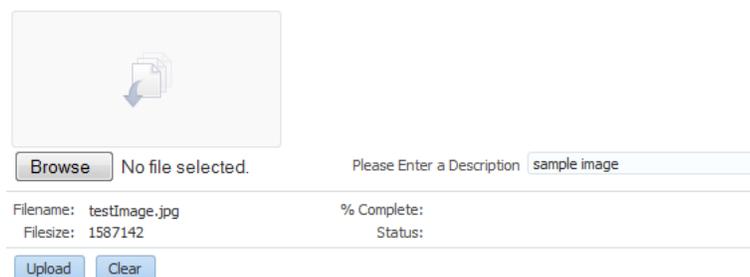
By default, the `inputFile` component allows upload of one file only, but it can be configured to upload multiple files. [Figure 11-48](#) shows the `inputFile` component configured to upload multiple files.

Figure 11-48 inputFile Component for Multiple Files

The user can select multiple files in the File Upload dialog that opens through the **Browse** button, or drag-and-drop multiple files in the drop section of the component. When files appear in the drop section, the user clicks **Upload** to upload the files, as shown in [Figure 11-49](#).

Figure 11-49 inputFile Component Showing Files Ready to Upload

Using Javascript APIs, you can configure the `inputFile` component to utilize custom user interface elements. For example, [Figure 11-50](#) shows the `inputFile` component with a description field that the user may use to enter a brief description of the selected file.

Figure 11-50 inputFile Component with Custom User Interface Element

For information about Javascript APIs that you can use to configure the `inputFile` component, see the documentation of class `AdfFileUploadManager` in *JavaScript API Reference for Oracle ADF Faces*.

The `inputFile` component can be placed in either an `h:form` tag or an `af:form` tag, but in either case, you have to set the form tag to support file upload. If you use the

JSF basic HTML `h:form`, set the `enctype` to `multipart/form-data`. This would make the request into a multipart request to support file uploading to the server. If you are using the ADF Faces `af:form` tag, set `usesUpload` to `true`, which performs the same function as setting `enctype` to `multipart/form-data` to support file upload.

The ADF Faces framework performs a generic upload of the file. You should create an `actionListener` or `action` method to process the file after it has been uploaded (for example, processing `xml` files, `pdf` files, and so on).

The value of an `inputFile` component is an instance of the `org.apache.myfaces.trinidad.model.UploadedFile` interface. Thus, when the file is uploaded, the value of the `inputFile` component becomes the instance of `org.apache.myfaces.trinidad.model.UploadedFile`. Therefore, if you need to access the value (that is, the file itself), you need to use the API for this interface. Accessing the component itself through the `binding` attribute only accesses the component and not the uploaded file.

 **Note:**

The API of the `org.apache.myfaces.trinidad.model.UploadedFile` interface lets you get at the actual byte stream of the file, as well as the file's name, its MIME type, and its size. The API does not allow you to get path information from the client about from where the file was uploaded.

The uploaded file may be stored as a file in the file system, but may also be stored in memory; the API hides that difference. The filter ensures that the `UploadedFile` content is cleaned up after the request is complete. Because of this, you cannot usefully cache `UploadedFile` objects across requests. If you need to keep the file, you must copy it into persistent storage before the request finishes.

For example, instead of storing the file, add a message stating the file upload was successful using a managed bean as a response to the `ValueChangeEvent` event, as shown in [Example 11-1](#).

You can also handle the upload by binding the value directly to a managed bean, as shown in [Example 11-2](#).

 **Note:**

If you are using the `inputFile` component to upload multiple files, note that the return type of `event.getNewValue()` is `List<UploadedFile>`, instead of `UploadedFile`. The value binding for the managed bean is also `List<UploadedFile>`, not `UploadedFile`.

Example 11-1 Using `valueChangeListener` to Display Upload Message

```
JSF Page Code ----->
<af:form usesUpload="true" id="f1">
  <af:inputFile label="Upload:"
    valueChangeListener="#{managedBean.fileUploaded}" id="if1"/>
  <af:button text="Begin" id="b1"/>
</af:form>
```

```
</af:form>

Managed Bean Code ---->
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import org.apache.myfaces.trinidad.model.UploadedFile;

public class ABackingBean
{
    ...
    public void fileUploaded(ValueChangeEvent event)
    {
        UploadedFile file = (UploadedFile) event.getNewValue();
        if (file != null)
        {
            FacesContext context = FacesContext.getCurrentInstance();
            FacesMessage message = new FacesMessage(
                "Successfully uploaded file " + file.getFilename() +
                " (" + file.getLength() + " bytes)");
            context.addMessage(event.getComponent().getClientId(context), message);
            // Here's where we could call file.getInputStream()
        }
    }
}
}
```

Example 11-2 Example 11-13 Binding the Value to a Managed Bean

```
JSF Page Code ---->
<af:form usesUpload="true">
    <af:inputFile label="Upload:" value="#{managedBean.file}" id="if1"/>
    <af:button text="Begin" action="#{managedBean.doUpload}" id="b1"/>
</af:form>

Managed Bean Code ---->
import org.apache.myfaces.trinidad.model.UploadedFile;public class AManagedBean
{
    public UploadedFile getFile()
    {
        return _file;
    }
    public void setFile(UploadedFile file)
    {
        _file = file;
    }

    public String doUpload()
    {
        UploadedFile file = getFile();
        // ... and process it in some way
    }
    private UploadedFile _file;
}
}
```

How to Use the inputFile Component

A Java class must be bound to the `inputFile` component. This class will be responsible for containing the value of the uploaded file.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using File Upload](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add an `inputFile` component:

1. Create a Java class that will hold the value of the input file. It must be an instance of the `org.apache.myfaces.trinidad.model.UploadedFile` interface.
2. Select the `af:form` component and set **UsesUpload** to true.
3. In the Components window, from the Text and Selection panel, drag and drop an **Input File** onto the page.
4. Set **value** to be the class created in Step 1.
5. If you want the value of the component to appear as read-only until the user hovers over it, expand the Appearance section and set **Editable** to `onAccess`. If you want the component to always appear editable, select `always`. If you want the value to be inherited from an ancestor component, select `inherit`.

 **Note:**

If you select `inherit`, and no ancestor components define the `editable` value, then the value `always` is used.

6. In the Components window, from the General Controls panel, drag and drop any command component onto the page. This will be used to initiate the upload process.
7. With the command component selected, set the `actionListener` attribute to a listener that will process the file after it has been uploaded.

How to Configure the `inputFile` Component to Upload Multiple Files

Use the `uploadType` and `maximumFiles` attributes to configure the `inputFile` component to upload multiple files.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using File Upload](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To configure an `inputFile` component to upload multiple files:

1. In the form, select the `inputFile` component.
2. In the Properties window, expand the Appearance section and set the following:
 - `autoHeightRows`: Specify the number of rows used to size the height of the `inputFile` component. The value must be lower than the value of `rows`.

- `rows`: Specify the number of files that will appear in the drop section. By default, it is set to 5.
3. Expand the Advanced section and set the `maximumFiles` attribute to specify the number of maximum files the user can upload. By default, it is set to 1 and allows upload of one file only. When set to less than 1 (for example, -1), it enables upload of unlimited number of files.
 4. Expand the Behavior section and set the `uploadType` attribute to specify whether the files would be uploaded automatically, or requires user to click **Upload** button to upload files.

Table 11-3 lists the possible values of the `uploadType` attribute.

Table 11-3 `uploadType` Values for the `inputFile` Component

Value	Description
<code>submit</code>	Upload one file only. The drop section, where the user can drag-and-drop files, is not displayed.
<code>auto</code>	Show the drop section and enable upload of multiple files. The upload starts immediately when the files appear in drop section. If <code>maximumFiles</code> is set to 1, the user can upload multiple files by selecting one file at a time, instead of selecting multiple files together.
<code>manual</code>	Show the drop section and enable upload of multiple files. The upload starts when the Upload button is clicked. If <code>maximumFiles</code> is set to 1, the user can upload multiple files by selecting one file at a time, instead of selecting multiple files together.
<code>autoIfMultiple</code>	Upload multiple files. The upload starts immediately when the files appear in the drop section. By default, <code>uploadType</code> is set to <code>autoIfMultiple</code> . If <code>maximumFiles</code> is set to 1, the user can select and upload one file only. The drop section is also not displayed.
<code>manualIfMultiple</code>	Upload multiple files. The upload starts when the Upload button is clicked. If <code>maximumFiles</code> is set to 1, the user can select and upload one file only. The drop section is also not displayed.

5. Expand the Other section and set the `displayMode` attribute to specify whether the multiple file upload user interface is displayed. By default, it is set to `default` and enables the display of multiple file upload user interface. The valid values are `default`, `dropArea`, and `none`.

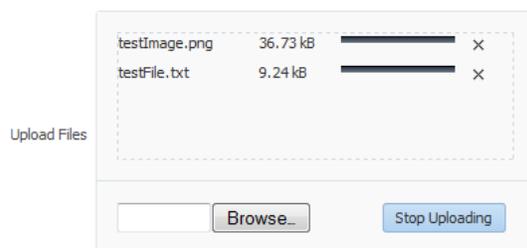
 **Note:**

Do not set `required` to `true` if `uploadType` is set to `auto` and `autoSubmit` is set to `true`, as it might throw validation errors.

See [What You May Need to Know About Customizing User Interface of inputFile Component](#).

To remove an uploaded file from the drop section, or cancel upload of a file that is being uploaded, click the **Cancel** icon next to the file name and the progress bar. To cancel upload of all files, click the **Stop Uploading** button, as shown in [Figure 11-51](#).

Figure 11-51 File Being Uploaded Using inputFile Component



What You May Need to Know About Temporary File Storage

Because ADF Faces will temporarily store files being uploaded (either on disk or in memory), by default it limits the size of acceptable incoming upload requests to avoid denial-of-service attacks that might attempt to fill a hard drive or flood memory with uploaded files. By default, only the first 100 kilobytes in any one request will be stored in memory. Once that has been filled, disk space will be used. Again, by default, that is limited to 2,000 kilobytes of disk storage for any one request for all files combined. Once these limits are exceeded, the filter will throw an `EOFException`.

Files are, by default, stored in the temporary directory used by the `java.io.File.createTempFile()` method, which is usually defined by the system property `java.io.tmpdir`. Obviously, this will be insufficient for some applications, so you can configure these values using three servlet context initialization parameters, as shown in the following example.

```
<context-param>
  <!-- Maximum memory per request (in bytes) -->
  <param-name>org.apache.myfaces.trinidad.UPLOAD_MAX_MEMORY</param-name>
  <!-- Use 500K -->
  <param-value>512000</param-value>
</context-param>
<context-param>
  <!-- Maximum disk space per request (in bytes) -->
  <param-name>org.apache.myfaces.trinidad.UPLOAD_MAX_DISK_SPACE</param-name>
  <!-- Use 5,000K -->
  <param-value>5120000</param-value>
</context-param>
<context-param>
  <!-- directory to store temporary files -->
  <param-name>org.apache.myfaces.trinidad.UPLOAD_TEMP_DIR</param-name>
  <!-- Use a TrinidadUploads subdirectory of /tmp -->
  <param-value>/tmp/TrinidadUploads/</param-value>
</context-param>
<context-param>
  <!-- Maximum file size that can be uploaded.-->
  <param-name>org.apache.myfaces.trinidad.UPLOAD_MAX_FILE_SIZE</param-name>
  <!-- Use 5,000K -->
```

```

    <param-value>5120000</param-value>
  </context-param>
  <!-- This filter is always required; one of its functions is
       file upload. -->
  <filter>
    <filter-name>trinidad</filter-name>
    <filter-class>org.apache.myfaces.trinidad.webapp.TrinidadFilter</filter-class>
  </filter>

```

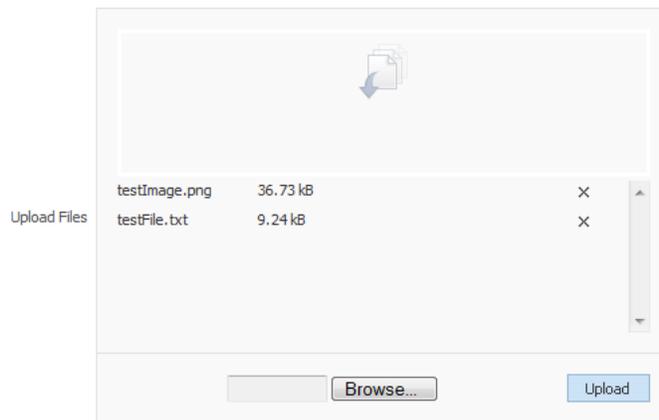
You can customize the file upload process by replacing the entire `org.apache.myfaces.trinidad.webapp.UploadedFileProcessor` class with the `<uploaded-file-processor>` element in the `trinidad-config.xml` configuration file. Replacing the `UploadedFileProcessor` class makes the parameters listed in the previous example irrelevant, they are processed only by the default `UploadedFileProcessor` class.

The `<uploaded-file-processor>` element must be the name of a class that implements the `oracle.adf.view.rich.webapp.UploadedFileProcessor` interface. This API is responsible for processing each individual uploaded file as it comes from the incoming request, and then making its contents available for the rest of the request. For most applications, the default `UploadedFileProcessor` class is sufficient, but applications that need to support uploading very large files may improve their performance by immediately storing files in their final destination, instead of requiring ADF Faces to handle temporary storage during the request.

What You May Need to Know About Uploading Multiple Files

The `inputFile` component uses HTML 5 to support the drag-and-drop functionality and upload of multiple files. In browsers that do not support HTML 5, a Java applet is used for drag-and-drop functionality and upload of multiple files, as shown in [Figure 11-52](#).

Figure 11-52 `inputFile` Component in a Non-HTML 5 Browser



If the browser does not support HTML5 and Java is also not available, then the drop section in the `inputFile` component is not displayed.

The `inputFile` component can only upload files that are smaller than 2 GB when in single file upload mode. In multiple file upload mode, the `inputFile` component can upload files greater than 2 GB, by default, by splitting them into chunks of 2 GB in size.

The chunk size can be controlled by the parameter `org.apache.myfaces.trinidad.UPLOAD_MAX_CHUNK_SIZE` in `web.xml` whose default and maximum value is 2 GB. For example:

```
<context-param>
  <!-- Maximum file chunk size that can be uploaded.-->
  <param-name>org.apache.myfaces.trinidad.UPLOAD_MAX_CHUNK_SIZE</param-name>
  <!-- Use 1,000 MB as chunk size -->
  <param-value>1000000000</param-value>
</context-param>
```

Note that not all browsers support the uploading of large files using the chunk functionality. For information about the browser certification, see the Certification Information page on the Oracle Technology Network at:

<http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html>

After uploading all files, you must ensure that the form is submitted, else the `inputFile` component data will not be uploaded to the server. If `autoSubmit` is set to `true` on the `inputFile` component, then the form is submitted automatically after all the files have finished uploading. After the form has been submitted, the `inputFile` component is refreshed and the file list of the drop section becomes empty so that more files can be uploaded. To show the list of uploaded files, add `ValueChangeListener` or bind the value to a managed bean, as described in [Example 11-1](#).

What You May Need to Know About Customizing User Interface of `inputFile` Component

You can customize the user interface of the `inputFile` component with the `displayMode` attribute. If the `displayMode` attribute is set to `none`, no user interface is displayed, and as a developer you will need to create a custom user interface to invoke the APIs to upload files. If the attribute is set to `dropArea`, a drop area is rendered that supports drag and drop of files, and you will be responsible for adding the rest of the custom user interface. The customized interface of the `inputFile` component is visible only when it is configured to upload multiple files.

[Figure 11-53](#) shows the `inputFile` component with different values for `displayMode`.

Figure 11-53 displayMode Attribute Values for inputFile ComponentinputFile component when
displayMode is set to *default*inputFile component when
displayMode is set to *dropArea*

Select File

inputFile component when
displayMode is set to *none*

The following example shows an `inputFile` component with `displayMode` set to `dropArea`, and customized to add input fields and buttons to browse and upload files. An additional `inputText` component is added to enter the description of the uploaded file.

```
<af:inputFile binding="#{editor.component}" id="bound1" immediate="true"
    maximumFiles="-1" valueChangeListener="#{demoFile.fileUpdate}"
    displayMode="dropArea" uploadType="manual" contentType="width:200px;
    height:100px" />

<af:panelFormLayout maxColumns="2" rows="1">
  <input type="file" id="dmoTpl:cidf1" />
  <af:inputText clientComponent="true" id="cidd1" label="Please Enter a
  Description" />
</af:panelFormLayout>

<af:panelFormLayout maxColumns="2" rows="2">
  <af:panelLabelAndMessage for="cidn1" label="Filename:" id="plm1">
    <af:outputText clientComponent="true" id="cidn1" />
  </af:panelLabelAndMessage>
  <af:panelLabelAndMessage for="cids1" label="Filesize:" id="plm2">
    <af:outputText clientComponent="true" id="cids1" />
  </af:panelLabelAndMessage>
  <af:panelLabelAndMessage for="cidp1" label="% Complete:" id="plm3">
    <af:outputText clientComponent="true" id="cidp1" />
  </af:panelLabelAndMessage>
  <af:panelLabelAndMessage for="cidst1" label="Status:" id="plm4">
    <af:outputText clientComponent="true" id="cidst1" />
  </af:panelLabelAndMessage>
</af:panelFormLayout>
```

```

</af:panelLabelAndMessage>
</af:panelFormLayout>

<af:panelFormLayout maxColumns="2" rows="1">
  <af:button id="cup1" text="Upload">
    <af:clientListener method="uploadClick" type="click"/>
  </af:button>
  <af:button id="cl1" text="Clear">
    <af:clientListener method="clearQueue" type="click"/>
  </af:button>
</af:panelFormLayout>

```

Figure 11-54 shows the `inputFile` component at runtime.

Figure 11-54 Customized inputFile Component



You can use the JavaScript API methods to customize the behavior of `inputFile` component. See the *JavaScript API Reference for Oracle ADF Faces*.

Using Code Editor

Using the code editor users can edit the code on the go during runtime. This is achieved using ADF `af:codeEditor` component.

The `af:codeEditor` component provides an in-browser code editing solution and enables the user to display and edit program code at runtime in the Fusion web application. The input field of the code editor component accepts text, and provides some common code editing functionalities such as a toolbar, syntactical color coding of keywords, code completion, basic validation, highlighting errors, and a message pane for logs. Using the code editor, the user won't need to run separate IDE software to test program code for errors or warnings.

The code editor component supports Javascript, XML, and Groovy languages, as shown in Figure 11-55.

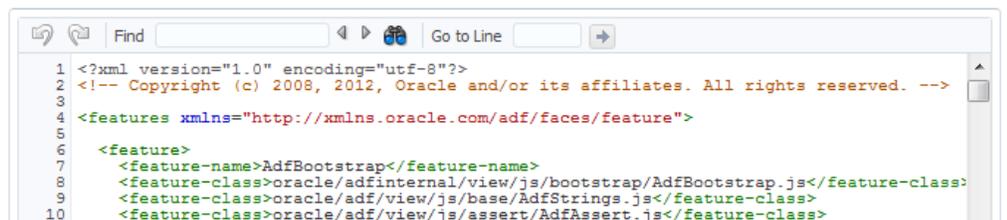
Figure 11-55 Code Editor Component using Javascript, XML, and Groovy

Javascript



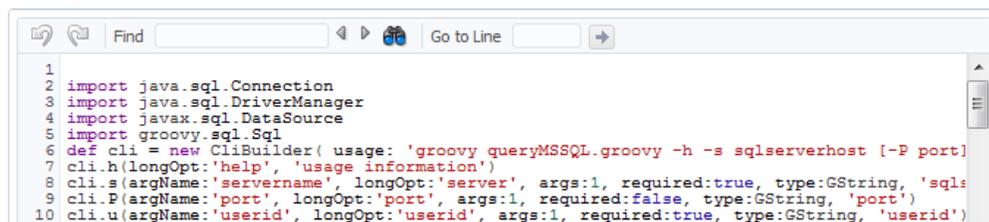
```
1 /** Copyright (c) 2008, 2012, Oracle and/or its affiliates. All rights reserved. */
2 // Utilities required for testFileExplorer.jspx
3
4 var TestFileUtils = new Object();
5
6 // Action listener for the unimplemented actions
7 TestFileUtils.notImplemented = function(event)
8 {
9     alert("Not yet implemented: " + event);
10 }
```

XML



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- Copyright (c) 2008, 2012, Oracle and/or its affiliates. All rights reserved. -->
3
4 <features xmlns="http://xmlns.oracle.com/adf/faces/feature">
5
6     <feature>
7         <feature-name>AdfBootstrap</feature-name>
8         <feature-class>oracle/adfinternal/view/js/bootstrap/AdfBootstrap.js</feature-class>
9         <feature-class>oracle/adf/view/js/base/AdfStrings.js</feature-class>
10        <feature-class>oracle/adf/view/js/assert/AdfAssert.js</feature-class>
11    </feature>
12 </features>
```

Groovy



```
1
2 import java.sql.Connection
3 import java.sql.DriverManager
4 import javax.sql.DataSource
5 import groovy.sql.Sql
6 def cli = new CliBuilder( usage: 'groovy queryMSSQL.groovy -h -s sqlserverhost [-P port]
7 cli.h(longOpt:'help', 'usage information')
8 cli.s(argName:'servername', longOpt:'server', args:1, required:true, type:GString, 'sql'
9 cli.P(argName:'port', longOpt:'port', args:1, required:false, type:GString, 'port')
10 cli.u(argName:'userid', longOpt:'userid', args:1, required:true, type:GString, 'userid')
```

The code editor component provides the following functionalities:

- Line numbering
- Undo and redo operations (also possible using keyboard shortcuts Ctrl+Z and Ctrl+Y)
- Jump to a specific line
- Find and replace
- Color-coded text
- Highlighting syntaxes and search terms
- Code completion
- Auto-indent
- Auto-format
- Message pane for error messages
- Support for large files with more than thousand lines of code

To add or edit code in code editor, the user can simply click in the editor area and start typing. Control+Spacebar displays a contextual list of hints for code completion. To

use other editing features, press F2 to enable edit mode. Once in edit mode, the user can use Tab to indent and Shift+Tab to backward-indent a line of code. With edit mode disabled, Tab and Shift+Tab provide navigation to the next and previous components.

The user can use the toolbar (shown in [Figure 11-56](#)) to undo and redo the changes, search and replace text, and jump to a specific line number.

Figure 11-56 Code Editor Toolbar



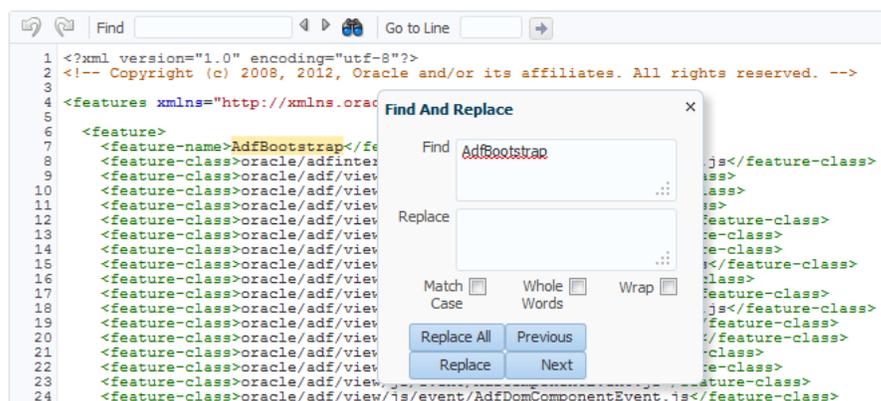
To search for a string, enter the search term in the **Find** field, and click **Find Next** or **Find Previous** icons to locate the search string in the code editor. [Figure 11-57](#) shows the Find field of the toolbar used to search a string in the code editor.

Figure 11-57 Using the Find Field of Code Editor Toolbar



To search a case sensitive string, or replace a search term, open the Find and Replace dialog from the **Find and Replace** icon, and perform the operations from the dialog, as shown in [Figure 11-58](#).

Figure 11-58 Using Find and Replace Dialog of Code Editor



 **Note:**

If the **Whole Words** checkbox is selected, the Find and Replace dialog cannot search for a non-English string in the editor. However, using the **Replace All** button, you can replace all instances of the non-English string while the **Whole Words** checkbox is selected.

To jump to a specific line number, enter the number in the **Go to Line** field and click **Jump to line**, as shown in [Figure 11-59](#).

Figure 11-59 Using Go To Line Feature of Code Editor



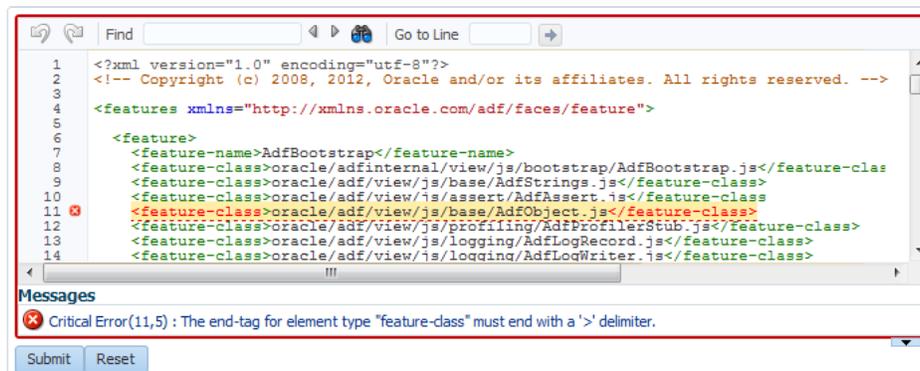
```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- Copyright (c) 2008, 2012, Oracle and/or its affiliates. All rights reserved. -->
3
4 <features xmlns="http://xmlns.oracle.com/adf/faces/feature">
5
6   <feature>
7     <feature-name>AdfBootstrap</feature-name>
8     <feature-class>oracle/adf/internal/view/js/bootstrap/AdfBootstrap.js</feature-class>
9     <feature-class>oracle/adf/view/js/base/AdfStrings.js</feature-class>
10    <feature-class>oracle/adf/view/js/assert/AdfAssert.js</feature-class>
11    <feature-class>oracle/adf/view/js/base/AdfObject.js</feature-class>
12    <feature-class>oracle/adf/view/js/profiling/AdfProfilerStub.js</feature-class>

```

The code editor component can be configured to list all warnings and errors in a message pane that is also provided with the code editor component. [Figure 11-60](#) shows the message pane listing all XML errors noticed by the XML parser running on the server.

Figure 11-60 Message Pane of Code Editor



```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- Copyright (c) 2008, 2012, Oracle and/or its affiliates. All rights reserved. -->
3
4 <features xmlns="http://xmlns.oracle.com/adf/faces/feature">
5
6   <feature>
7     <feature-name>AdfBootstrap</feature-name>
8     <feature-class>oracle/adf/internal/view/js/bootstrap/AdfBootstrap.js</feature-cla
9     <feature-class>oracle/adf/view/js/base/AdfStrings.js</feature-class>
10    <feature-class>oracle/adf/view/js/assert/AdfAssert.js</feature-class>
11    <feature-class>oracle/adf/view/js/base/AdfObject.js</feature-class>
12    <feature-class>oracle/adf/view/js/profiling/AdfProfilerStub.js</feature-class>
13    <feature-class>oracle/adf/view/js/logging/AdfLogRecord.js</feature-class>
14    <feature-class>oracle/adf/view/js/logging/AdfLogWriter.js</feature-class>

```

Messages

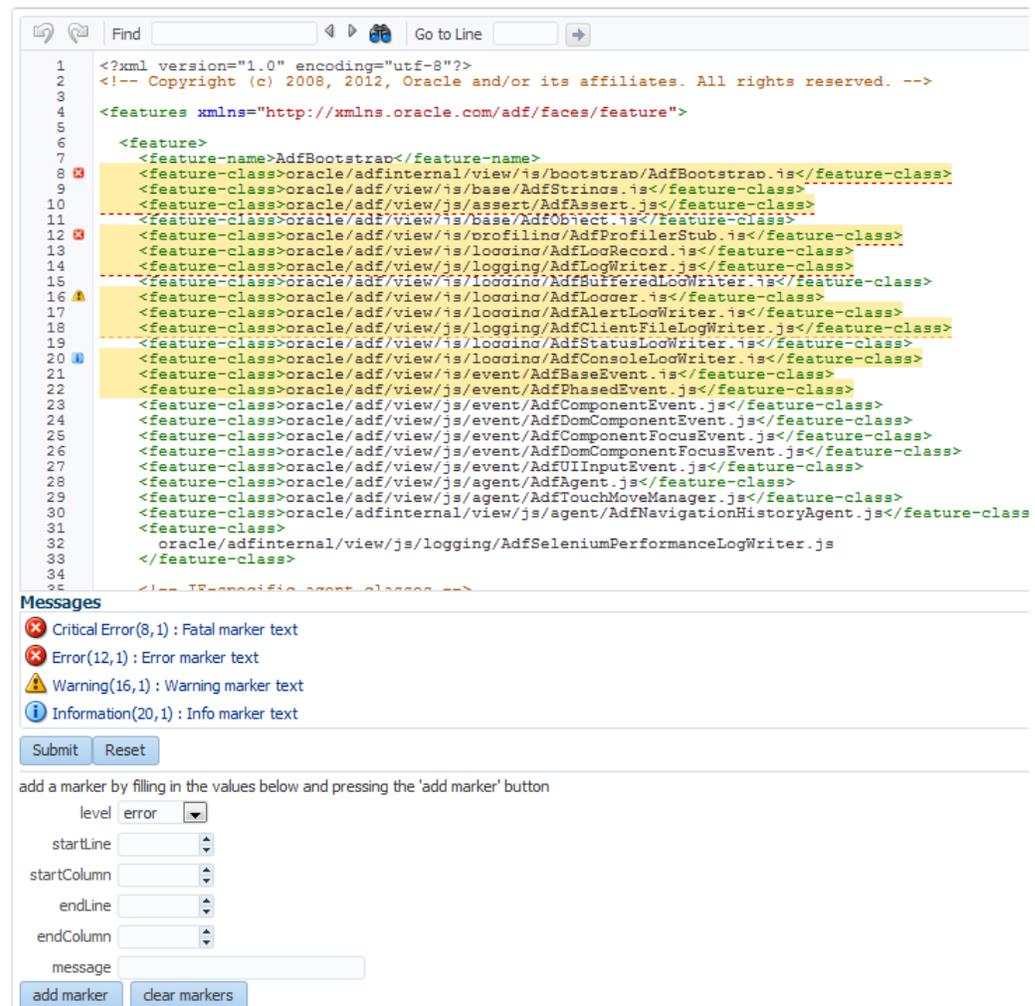
 Critical Error(11,5) : The end-tag for element type "feature-class" must end with a '>' delimiter.

Submit Reset

The message pane is a non-editable region that resides below the text area of the code editor. It is used to display code-related status information, such as validation support for code compilation, and any error or warning messages. Clicking a message in the message pane navigates you to the respective code line in the code editor.

You can also configure the code editor to programmatically add various types of markers. [Figure 11-61](#) shows the code editor with critical error, error, warning, and information markers.

Figure 11-61 Using Markers in Code Editor



How to Add a codeEditor Component

When you add a `codeEditor` component, use the `language` attribute to configure the programming language used by the code editor.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Code Editor](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Input Components and Forms](#).

To add a `codeEditor` component:

1. In the Components window, from the Text and Selection panel, drag and drop a Code Editor component onto the page.
2. In the Properties window, expand the Common section, and set **Language**. The valid values are `javascript`, `groovy`, and `xml`.

3. Expand the Appearance section, and set the following:
 - **LineNumbers:** Specify whether line numbers should be visible in the code editor.
The valid values are `yes` and `no`.
 - **Simple:** Set to `true` if you do not want the label to be displayed.
4. Expand the Behavior section, and set the following
 - **ReadOnly:** Specify whether the code in the code editor can be edited or displayed as output-style text.
 - **Disabled:** Specify whether or not the code editor should be disabled.

12

Using Tables, Trees, and Other Collection-Based Components

This chapter describes how to display structured data in components that can iterate through collections of data and then display each row in the collection, using the ADF Faces `table`, `tree` and `treeTable`, `listView`, and `carousel` components. If your application uses the Fusion technology stack, then you can use data controls to create these components. See *Creating ADF Databound Tables, Displaying Master-Detail Data, and Using More Complex Databound ADF Faces Components* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About Collection-Based Components](#)
- [Common Functionality in Collection-Based Components](#)
- [Displaying Data in Tables](#)
- [Adding Hidden Capabilities to a Table](#)
- [Enabling Filtering in Tables](#)
- [Displaying Data in Trees](#)
- [Displaying Data in Tree Tables](#)
- [Passing a Row as a Value](#)
- [Displaying Table Menus, Toolbars, and Status Bars](#)
- [Displaying a Collection in a List](#)
- [Displaying Images in a Carousel](#)
- [Exporting Data from Table, Tree, or Tree Table](#)
- [Accessing Selected Values on the Client from Collection-Based Components](#)

About Collection-Based Components

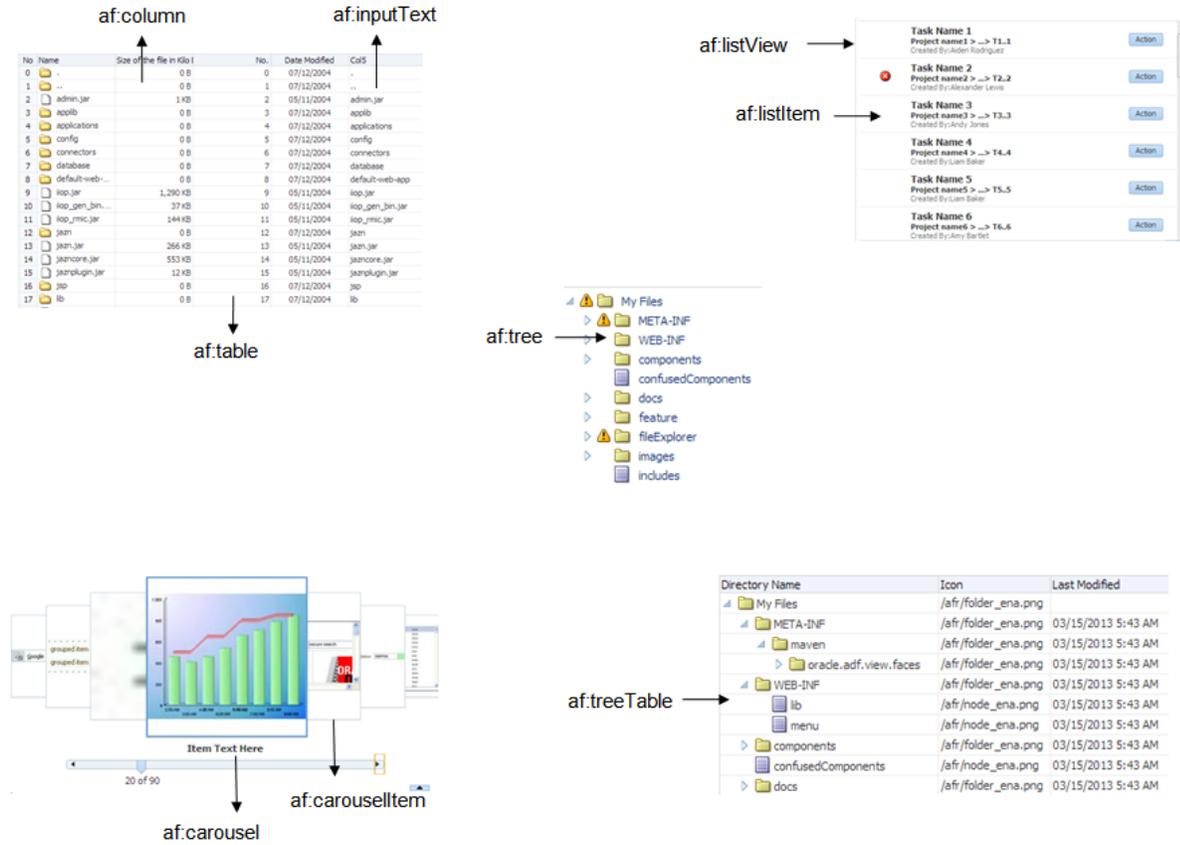
The basic understanding of what is collection-based components is explained. The additional functionalities, such as Customizing the toolbar, Active Data and others, also can be used along with ADF table and tree components.

ADF Faces provides components that you can use to iterate through and display collections of structured data. Instead of containing a child component for each record to be displayed, and then binding these components to the individual records, these components are bound to a complete collection, and they then repeatedly render one component (for example an `outputText` component), by stamping the value for each record.

Structured data can be displayed as a simple table consisting of a number of rows and one column, using the `Listview` component, or multiple columns using the ADF Faces `table` component. Hierarchical data can be displayed either as tree structures using

ADF Faces tree component, or in a table format, using ADF Faces tree table component. A collection of images can be displayed in a carousel component. Figure 12-1 shows the ADF Faces collection-based components.

Figure 12-1 ADF Faces Collection-Based Components



 **Tip:**

When you do not want to use a table, but still need the same stamping capabilities, you can use the iterator tag. For example, say you want to display a list of periodic table elements, and for each element, you want to display the name, atomic number, symbol, and group. You can use the iterator tag as shown in the following example.

```
<af:iterator var="row" first="3" rows="3" varStatus="stat"
  value="#{periodicTable.tableData}" >
  <af:outputText value="#{stat.count}.Index:#{stat.index} of
    #{stat.model.rowCount}"/>
  <af:outputText label="Element Name" value="#{row.name}"/>
  <af:outputText label="Atomic Number" value="#{row.number}"/>
  <af:outputText label="Symbol" value="#{row.symbol}"/>
  <af:outputText label="Group" value="#{row.group}"/>
</af:iterator>
```

Each child is stamped as many times as necessary. Iteration starts at the index specified by the `first` attribute for as many indexes specified by the `row` attribute. If the `row` attribute is set to 0, then the iteration continues until there are no more elements in the underlying data.

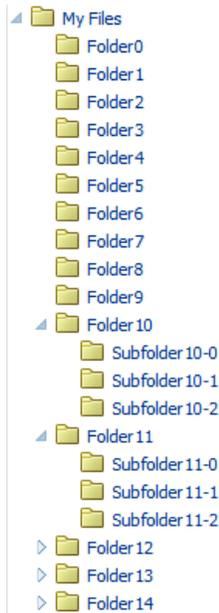
Collection-Based Component Use Cases and Examples

Collection-based components are used to display structured information. For example, as shown in [Figure 12-2](#), the Table tab in the File Explorer application uses a table to display the contents of the selected directory.

Figure 12-2 Table Component in the File Explorer Application

Name	Size (KB)	Type	Date Modified	Properties
 File2.doc	10	Document File	06/18/2009 6:06 PM	Properties
 File2.html	10	HTML File	06/18/2009 6:06 PM	Properties
 File2.pdf	10	PDF File	06/18/2009 6:06 PM	Properties
 File2.xls	10	XLS File	06/18/2009 6:06 PM	Properties

Hierarchical data (that is data that has parent/child relationships), such as the directory in the File Explorer application, can be displayed as expandable trees using the tree component. Items are displayed as nodes that mirror the parent/child structure of the data. Each top-level node can be expanded to display any child nodes, which in turn can also be expanded to display any of their child nodes. Each expanded node can then be collapsed to hide child nodes. [Figure 12-3](#) shows the file directory in the File Explorer application, which is displayed using a tree component.

Figure 12-3 Tree Component in the File Explorer Application

Hierarchical data can also be displayed using tree table components. The tree table also displays parent/child nodes that are expandable and collapsible, but in a tabular format, which allows the page to display attribute values for the nodes as columns of data. For example, along with displaying a directory's contents using a table component, the File Explorer application has another tab that uses the tree table component to display the contents, as shown in [Figure 12-4](#).

Figure 12-4 Tree Table in the File Explorer Application

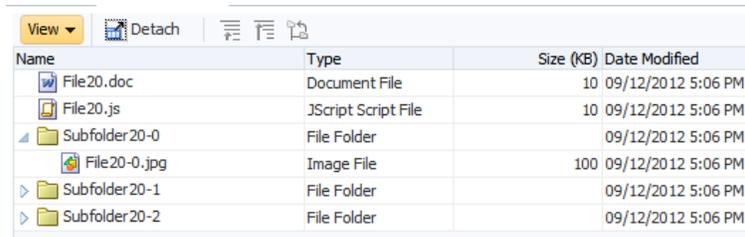
Name	Type	Size (KB)	Date Modified
File 11.doc	Document File	10	04/25/2013 5:57 PM
File 11.js	JScript Script File	10	04/25/2013 5:57 PM
Subfolder 11-0	File Folder		04/25/2013 5:57 PM
File 11-0.jpg	Image File	100	04/25/2013 5:57 PM
Subfolder 11-1	File Folder		04/25/2013 5:57 PM
File 11-1.jpg	Image File	100	04/25/2013 5:57 PM
Subfolder 11-2	File Folder		04/25/2013 5:57 PM

Like the tree component, the tree table component can show the parent/child relationship between items. And like the table component, the tree table component can also show any attribute values for those items in a column. Most of the features available on a table component are also available in tree table component.

You can add a toolbar and a status bar to tables, trees, and tree tables by surrounding them with the `panelCollection` component. The top panel contains a standard menu bar as well as a toolbar that holds menu-type components such as menus and menu options, toolbars and toolbar buttons, and status bars. Some buttons and menus are added by default. For example, when you surround a table, tree, or tree table with a `panelCollection` component, a toolbar that contains the **View** menu is added. This menu contains menu items that are specific to the table, tree, or tree table component.

Figure 12-5 shows the tree table from the File Explorer application with the toolbar, menus, and toolbar buttons created using the `panelCollection` component.

Figure 12-5 TreeTable with Panel Collection



Name	Type	Size (KB)	Date Modified
File20.doc	Document File	10	09/12/2012 5:06 PM
File20.js	JScript Script File	10	09/12/2012 5:06 PM
Subfolder20-0	File Folder		09/12/2012 5:06 PM
File20-0.jpg	Image File	100	09/12/2012 5:06 PM
Subfolder20-1	File Folder		09/12/2012 5:06 PM
Subfolder20-2	File Folder		09/12/2012 5:06 PM

The `listView` component is a light-weight table that allows you to display structured data in a list format. Unlike a table, it does not have columns, which allows you to easily present data in a variety of patterns, beyond a simple tabular layout.

The components that display the actual data are contained in a single child `listItem` component. Figure 12-6 shows a `listView` component that contains one child `listItem` component. The `listItem` component contains a mix of layout components, output components and button components.

Figure 12-6 The listView Component Uses listItem Components to Hold Data for Each Row



Task Name 1 Project name1 > ...> T1..1 Created By:Annett Barnes	Action
 Task Name 2 Project name2 > ...> T2..2 Created By:Benjamin Lee	Action
Task Name 3 Project name3 > ...> T3..3 Created By:Amelia Sanchez	Action
Task Name 4 Project name4 > ...> T4..4 Created By:Jacob Miller	Action

The `listView` component can also display hierarchical data. When a component that is bound to the parent data is placed in the `groupHeaderStamp` facet, that data is displayed in a header. Figure 12-7 shows how the alphabet letters, which are the parent data, are displayed in headers, while the child personnel data is displayed in rows below the parent.

Figure 12-7 Hierarchical Data Can be Displayed in Groups

The carousel component displays a collection of images in a revolving carousel, as shown in [Figure 12-8](#). Users can change the image at the front either by using the slider at the bottom or by clicking one of the auxiliary images to bring that specific image to the front.

Figure 12-8 The ADF Faces Carousel

Additional Functionality for Collection-Based Components

You may find it helpful to understand other ADF Faces features before you implement your collection-based components. Additionally, once you have added a collection-based component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that table and tree components can use.

- **Customizing the toolbar:** You can customize the toolbar included in the `panelCollection` component, which provides menus, toolbars, and status bars for the table and tree table components. For information about menus, toolbars, and toolbar buttons, see [Using Menus, Toolbars, and Toolboxes](#).
- **Geometry management of the table width:** If the table is a child to a component that stretches its children, then this width setting will be overridden and the table will automatically stretch to fit its container. For information about how components stretch, see [Geometry Management and Component Stretching](#).
- **Active data:** If your application uses active data, then you can have the data in your tables and trees update automatically, whenever the data in the data source

changes. See *Using the Active Data Service in Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you wish to use active data, and your application uses ADF Business Components, then your tables must conform to the following:

- The table or tree is bound to homogeneous data which contains only a single attribute.
- The table does not use filtering.
- The tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags.

- **Events:** Collection-based components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- **Partial page rendering:** You may want a collection-based component to refresh to show new data based on an action taken on another component on the page. See [Using the Optimized Lifecycle](#).
- **Personalization:** Users can change the way the component displays at runtime (for example the user can reorder columns or change column widths), those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- **Accessibility:** You can make your components accessible. See [Developing Accessible ADF Faces Pages](#).
- **Automatic data binding:** If your application uses the Fusion technology stack, then you can create automatically bound tables and trees based on how your ADF Business Components are configured. See *Creating ADF Databound Tables and Displaying Master-Detail Data in Developing Web User Interfaces with Oracle ADF Faces*.

Common Functionality in Collection-Based Components

Using the functionalities available with collection-based components you can perform many functionalities. Some of them are, display data in a row in an ADF table structure and in node in an ADF tree structure, or you can decide to deliver either one row or multiple rows of content to the client machine.

Collection-based component share many of the same functionality, such as how data is delivered and how data can be displayed and edited. It is important that you understand this shared functionality and how it is configured before you use these components.

Displaying Data in Rows and Nodes

Instead of containing a child component for each record to be displayed, and then binding these components to the individual records, collection-based components are bound to a complete collection, and they then repeatedly render one component (for

example an `outputText` component) by stamping the value for each record. For example, say a table contains two child column components. Each column displays a single attribute value for the row using an output component and there are four records to be displayed. Instead of binding four sets of two output components to display the data, the table itself is bound to the collection of all four records and simply stamps one set of the output components four times. As each row is stamped, the data for the current row is copied into the `var` attribute on the table, from which the output component can retrieve the correct values for the row. For information about how stamping works, especially with client components, see [Accessing Client Collection Components](#).

The following example shows the JSF code for a table whose value for the `var` attribute is `row`. Each `outputText` component in a column displays the data for the row because its value is bound to a specific property on the variable.

```
<af:table var="row" value="#{myBean.allEmployees}">
  <af:column>
    <af:outputText value="#{row.firstname}"/>
  </af:column>
  <af:column>
    af:outputText value="#{row.lastname}"/>
  </af:column>
</af:table>
```

Collection-based components uses a `CollectionModel` class to access the data in the underlying collection. This class extends the JSF `DataModel` class and adds on support for row keys and sorting. In the `DataModel` class, rows are identified entirely by index. This can cause problems when the underlying data changes from one request to the next, for example a user request to delete one row may delete a different row when another user adds a row. To work around this, the `CollectionModel` class is based on row keys instead of indexes.

You may also use other model classes, such as `java.util.List`, `array`, and `javax.faces.model.DataModel`. If you use one of these other classes, the table component automatically converts the instance into a `CollectionModel` class, but without the additional functionality. For information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create tables and the collection model will be created for you. See *Creating ADF Databound Tables* in *Developing Web User Interfaces with Oracle ADF Faces*.

Content Delivery

The collection components are virtualized, meaning not all the rows that are there for the component on the server are delivered to and displayed on the client. You configure collection components to fetch a certain number of rows at a time from your data source. The data can be delivered to the components immediately upon rendering, when it is available, or lazily fetched after the shell of the component has

been rendered (by default, the components fetch data when it is available), as specified by the setting the `ContentDelivery` attribute on the component.

With immediate delivery, the data is fetched during the initial request. With **lazy delivery**, when a page contains one or more collection components, the page initially goes through the standard lifecycle. However, instead of fetching the data during that initial request, a special separate partial page rendering (PPR) request is run, and the number of rows set as the value of the fetch size for the component is then returned. Because the page has just been rendered, only the Render Response phase executes for the components, allowing the corresponding data to be fetched and displayed. When a user's actions cause a subsequent data fetch (for example scrolling in a table for another set of rows), another PPR request is executed.

When content delivery is configured to be delivered when it is available, the framework checks for data availability during the initial request, and if it is available, it sends the data to the component and inlines it for rendering. If it is not available, the data is loaded during the separate PPR request, as it is with lazy delivery. Note that in JDeveloper releases prior to 12.2.1.0, two separate requests are required to send the data and to inline the data upon rendering the component. If you prefer the content delivery performance of an earlier release, you can set the `ContentDelivery` attribute to `lazy`; this will force inlining of the data only during a subsequent.

 **Note:**

If your application *does not* use the Fusion technology stack, then you must explicitly add support for `whenAvailable` to your `CollectionModel` implementation. For an example, see the `WhenAvailableData.java` managed bean in the Faces demo application.

If your application *does* use the Fusion technology stack, then the `CollectionModel` implementation created for you automatically uses these APIs.

 **Performance Tip:**

Lazy delivery should be used when a data fetch is expected to be an expensive (slow) operation, for example, slow, high-latency database connection, or fetching data from slow nondatabase data sources like web services. Lazy delivery should also be used when the page contains a number of components other than a table, tree, or tree table. Doing so allows the initial page layout and other components to be rendered first before the data is available.

Immediate delivery should be used if the table, tree, or tree table is the only context on the page, or if the component is not expected to return a large set of data. In this case, response time will be faster than using lazy delivery (or in some cases, simply perceived as faster), as the second request will not go to the server, providing a faster user response time and better server CPU utilizations. Note however that only the number of rows configured to be the fetch block will be initially returned. As with lazy delivery, when a user's actions cause a subsequent data fetch, the next set of rows are delivered.

When available delivery provides the additional flexibility of using immediate when data is available during initial rendering or falling back on lazy when data is not initially available. Starting in the JDeveloper 12.2.1.0 release, when available is the default delivery content delivery mode.

The number of rows that are displayed on the client are just enough to fill the page as it is displayed in the browser. More rows are fetched as the user scrolls the component vertically (or if configured to page instead of scroll, when the user navigates to another set of rows). The `fetchSize` attribute determines the number of rows requested from the client to the server on each attempt to fill the component. For a table, the default value is 25. So if the height of the table is small, the fetch size of 25 is sufficient to fill the component. However, if the height of the component is large, there might be multiple requests for the data from the server. Therefore, the `fetchSize` attribute should be set to a higher number. For example, if the height of the table is 600 pixels and the height of each row is 18 pixels, you will need at least 45 rows to fill the table. With a `fetchSize` of 25, the table has to execute two requests to the server to fill the table. For this example, you would set the fetch size to 50.

However, if you set the fetch size too high, it will impact both server and client. The server will fetch more rows from the data source than needed and this will increase time and memory usage. On the client side, it will take longer to process those rows and attach them to the component.

 **Performance Tip:**

Reduce `fetchSize` whenever possible.

By default, on a desktop device, tables render a scroll bar that allows the users to scroll through the rows of data. Instead, when you want to configure the table to be paginated, you can set the `scrollPolicy` attribute to `page`. A paginated table displays a footer that allows the user to jump to specific pages of rows, as shown in [Figure 12-9](#).

Figure 12-9 Paginated Table

No	Name	Size of the file in Kilo B	No.	Date Modified
0	.	0 B	0	07/12/2004
1	..	0 B	1	07/12/2004
2	admin.jar	1 KB	2	05/11/2004
3	applib	0 B	3	07/12/2004
4	applications	0 B	4	07/12/2004
5	config	0 B	5	07/12/2004
6	connectors	0 B	6	07/12/2004
7	database	0 B	7	07/12/2004
8	default-web-...	0 B	8	07/12/2004
9	iiop.jar	1,290 KB	9	05/11/2004
10	iiop_gen_bin...	37 KB	10	05/11/2004
11	iiop_rmic.jar	144 KB	11	05/11/2004
12	jazn	0 B	12	07/12/2004
13	jazn.jar	266 KB	13	05/11/2004
14	jazncore.jar	553 KB	14	05/11/2004
15	jaznplugin.jar	12 KB	15	05/11/2004
16	jsp	0 B	16	07/12/2004
17	lib	0 B	17	07/12/2004
18	loadbalancer...	1 KB	18	05/11/2004
19	log	0 B	19	07/12/2004

Page 1 of 270 (1-20 of 5400 items) | 1 2 3 4 5 ... 270

When the viewport is too narrow to display the complete footer, the table displays a compact footer that shows only the page currently displayed and the navigation buttons, as shown in [Figure 12-10](#).

Figure 12-10 Paginated Table in Compact Mode

No	Name	Size of the file in Kilo B	No.	Date
0	.	0 B	0	07/12/2004
1	..	0 B	1	07/12/2004
2	admin.jar	1 KB	2	05/11/2004
3	applib	0 B	3	07/12/2004
4	applications	0 B	4	07/12/2004
5	config	0 B	5	07/12/2004
6	connectors	0 B	6	07/12/2004
7	database	0 B	7	07/12/2004
8	default-web-...	0 B	8	07/12/2004
9	iiop.jar	1,290 KB	9	05/11/2004
10	iiop_gen_bin...	37 KB	10	05/11/2004
11	iiop_rmic.jar	144 KB	11	05/11/2004
12	jazn	0 B	12	07/12/2004
13	jazn.jar	266 KB	13	05/11/2004
14	jazncore.jar	553 KB	14	05/11/2004
15	jaznplugin.jar	12 KB	15	05/11/2004
16	jsp	0 B	16	07/12/2004
17	lib	0 B	17	07/12/2004
18	loadbalancer...	1 KB	18	05/11/2004
19	log	0 B	19	07/12/2004

Page 1 of 270 | 1 2 3 4 5 ... 270

 **Note:**

By default, on tablet devices, tables are rendered to display the table as paginated and therefore do not display a scroll bar. If instead, you want to enable scroll bars so the table automatically loads the next set of rows when the user scrolls to the bottom of the table, you can set the `scrollPolicy` attribute to `scroll`. This option on tablets results in a behavior called implicit high-water mark scrolling and closely resembles the way virtualized touch scrolling (scrolling in both horizontal and vertical directions) behaves on tablet devices. Regular virtualized scrolling which results from setting `scrollPolicy` to `scroll` on a desktop machine is not implemented on tablets due to performance problems with virtualized scrolling of table data on tablets. You can, however, customize the caching behavior of implicit high-water mark scrolling by specifying the number of rows to cache to minimize database round-trips by setting a value for the `maxClientRows` attribute.

As with a table configured to scroll, the number of rows on a page is determined by the `fetchSize` attribute.

 **Note:**

By default, if you configure your tables to use scroll bars, on iOS operating systems, the scroll bars appear only when you mouseover the content. You can configure your application so that this same behavior occurs on other operating systems as well, by setting the `oracle.adf.view.rich.table.scrollbarBehavior` parameter in the `web.xml` file. See [Scrollbar Behavior in Tables](#).

 **Note:**

You can hide the scroll bar using the `-tr-overflow-style: autohiding-scrollbar` skinning property. For example:

```
af|table {  
  -tr-overflow-style: autohiding-scrollbar  
}
```

For information about skins, see [Customizing the Appearance Using Styles and Skins](#).

You can also configure the set of data that will be initially displayed using the `displayRow` attribute. By default, the first record in the data source is displayed in the top row or node and the subsequent records are displayed in the following rows or nodes. You can also configure the component to first display the last record in the source instead. In this case, the last record is displayed in the bottom row or node of the component, and the user can scroll up to view the preceding records. Additionally, you can configure the component to display the selected row. This can be useful if the user is navigating to the component, and based on some parameter, a particular row

will be programmatically selected. When configured to display the selected row, that row will be displayed at the top of the table and the user can scroll up or down to view other rows.

 **Note:**

You cannot use JavaScript to dynamically size a table or tree. The height of tables, trees and treetables is set the first time they are rendered and cannot be changed using JavaScript APIs.

Row Selection

You can configure selection to be either for no rows, for a single row, or for multiple rows using the `rowSelection` attribute (the `carousel` component does not allow multiple row selection). This setting allows you to execute logic against the selected rows. For example, you may want users to be able to select a row in a table or a node in a tree, and then to click a button that navigates to another page where the data for the selected row is displayed and the user can edit it.

 **Note:**

If you configure your component to allow multiple selection, users can select one row and then press the shift key to select another row, and all the rows in between will be selected. This selection will be retained even if the selection is across multiple data fetch blocks. Similarly, you can use the Ctrl key to select rows that are not next to each other.

For example, if you configure your table to fetch only 25 rows at a time, but the user selects 100 rows, the framework is able to keep track of the selection.

When the selected row (or node) of a component changes, the component triggers a selection event. This event reports which rows were just deselected and which rows were just selected. While the components handle selection declaratively, if you want to perform some logic on the selected rows, you need to implement code that can access those rows and then perform the logic. You can do this in a selection listener method on a managed bean. See [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).

 **Performance Tip:**

Users can navigate through the table using a mouse and the scrollbar, or using the up and down arrow keyboard keys. By default, a selection event is immediately fired when the user clicks a row. If the user is navigating through the rows using the arrow keys, this means that a selection event will be fired for each row, as the user navigates.

If you expect users to navigate through the table using the keys, you can set the `delaySelectionEvent` attribute to `true`, so that there is a 300 millisecond delay before the selection event is fired. If the user navigates to another row within the 300 milliseconds, the selection event is canceled.

Editing Data in Tables, Trees, and Tree Tables

You can choose the component used to display the actual data in a table, tree, or tree table. For example, you may want the data to be read-only, and therefore you might use an `outputText` component to display the data. Conversely, if you want the data to be able to be edited, you might use an `inputText` component, or if choosing from a list, one of the `SelectOne` components. All of these components are placed as children to the column component (in the case of a table and tree table) or within the `nodeStamp` facet (for a tree).

When you decide to use components whose value can be edited to display your data, you have the option of having the table, tree, or tree table either display all rows as available for editing at once, or display all but the currently active row as read-only using the `editingMode` attribute. For example, [Figure 12-11](#) shows a table whose rows can all be edited. The page renders using the components that were added to the page (for example, `inputText`, `inputDate`, and `inputComboBoxListOfValues` components).

Figure 12-11 Table Whose Rows Can All Be Edited

Number	Name	Size	Date Modified	A Spinbox	<input type="checkbox"/> select all	inputColor
0	 .	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
1	 ..	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
2	 admin.jar	1 KB	5/11/2004	 1979	<input type="checkbox"/> Cool	99FF99 
3	 applib	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
4	 applications	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
5	 config	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
6	 connectors	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 
7	 database	0 B	7/12/2004	 1979	<input type="checkbox"/> Cool	99FF99 

[Figure 12-12](#) shows the same table (that is, it uses `inputText`, `inputDate`, and `inputComboBoxListOfValues` components to display the data), but configured so that only the active row displays the editable components, by setting `editingMode` to `clickToEdit`. Users can then click on another row to make it editable (only one row is editable at a time). Note that `outputText` components are used to display the data in the noneditable rows, even though the same input components as in [Figure 12-11](#)

were used to build the page. The only row that actually renders those components is the active row.

 **Note:**

When you set `editingMode` to `readOnly`, all the components in the table will be rendered as read only, including input components. Also when the entire table is selected, the `editingMode` will be set to `readOnly`.

Figure 12-12 Table Allows Only One Row to Be Edited at a Time

Name	inputText	* Required field	inputComboBoxListOf	inputDate
 ojspc.jar	1 KB	05/11/2004		5/11/2004
 persistence	0 B	07/12/2004		7/12/2004
 rmic.jar	1 KB	05/11/2004		5/11/2004
 sql	0 B	07/12/2004		7/12/2004
 .	0 B	07/12/2004		7/12/2004
 ..	0 B	07/12/2004		7/12/2004
 admin.jar	1 KB	05/11/2004	<input type="text" value="v"/>	5/11/2004 
 applib	0 B	07/12/2004		7/12/2004
 applications	0 B	07/12/2004		7/12/2004
 config	0 B	07/12/2004		7/12/2004
 connectors	0 B	07/12/2004		7/12/2004
 database	0 B	07/12/2004		7/12/2004
 default-web-...	0 B	07/12/2004		7/12/2004

The currently active row is determined by the `activeRowKey` attribute on the table. By default, the value of this attribute is the first visible row of the table. When the table (or tree or tree table) is refreshed, that component scrolls to bring the active row into view, if it is not already visible. When the user clicks on a row to edit its contents, that row becomes the active row.

When you allow only a single row (or node) to be edited, the table (or tree or tree table) performs PPR when the user moves from one row (or node) to the next, thereby submitting the data (and validating that data) one row at a time. When you allow all rows to be edited, data is submitted whenever there is an event that causes PPR to typically occur, for example scrolling beyond the currently displayed rows or nodes.

 **Note:**

Trees and tables support browser copy and paste. When you mouse over a text field that can be copied, the cursor displays an I-bar. When you click the mouse to copy the text, that row becomes selected.

Not all editable components make sense to be displayed in a click-to-edit mode. For example, those that display multiple lines of HTML input elements may not be good candidates. These components include:

- `SelectManyCheckbox`
- `SelectManyListBox`

- `SelectOneListBox`
- `SelectOneRadio`
- `SelectManyShuttle`

 **Performance Tip:**

For increased performance during both rendering and postback, you should configure your table to allow editing only to a single row.

When you elect to allow only a single row to be edited at a time, the page will be displayed more quickly, as output components tend to generate less HTML than input components. Additionally, client components are not created for the read-only rows. Because the table (or tree, or tree table) performs PPR as the user moves from one row to the next, only that row's data is submitted, resulting in better performance than a table that allows all cells to be edited, which submits all the data for all the rows in the table at the same time. Allowing only a single row to be edited also provides more intuitive validation, because only a single row's data is submitted for validation, and therefore only errors for that row are displayed.

Using Popup Dialogs in Tables, Trees, and Tree Tables

You can configure your table, tree, or tree table so that popup dialogs will be displayed based on a user's actions. For example, you can configure a popup dialog to display some data from the selected row when the user hovers the mouse over a cell or node. You can also create popup context menus for when a user right-clicks a row in a table or tree table, or a node in a tree. Additionally, for tables and tree tables, you can create a context menu for when a user right-clicks anywhere within the table, but not on a specific row.

Tables, trees, and tree tables all contain the `contextMenu` facet. You place your popup context menu within this facet, and the associated menu will be displayed when the user right-clicks a row. When the context menu is being fetched on the server, the components automatically establish the currency to the row for which the context menu is being displayed. Establishing currency means that the current row in the model for the table now points to the row for which the context menu is being displayed. In order for this to happen, the `popup` component containing the menu must have its `contentDelivery` attribute set to `lazyUncached` so that the menu is fetched every time it is displayed.

 **Tip:**

If you want the context menu to dynamically display content based on the selected row, set the popup content delivery to `lazyUncached` and add a `setPropertyListener` tag to a method on a managed bean that can get the current row and then display data based on the current row:

```
<af:tree value="#{fs.treeModel}"
  contextMenuSelect="false" var="node" ..>
  <f:facet name="contextMenu">
    <af:popup id="myPopup" contentDelivery="lazyUncached">
      <af:setPropertyListener from="#{fs.treeModel.rowData}"
        to="#{dynamicContextMenuTable.currentTreeRowData}"
        type="popupFetch" />
      <af:menu>
        <af:menu text="Node Info (Dynamic)">
          <af:commandMenuItem actionListener=
            "#{dynamicContextMenuTable.alertTreeRowData}"
            text=
              "Name - #{dynamicContextMenuTable.currentTreeRowData.name}" />
          <af:commandMenuItem actionListener=
            "#{dynamicContextMenuTable.alertTreeRowData}"
            text=
              "Path - #{dynamicContextMenuTable.currentTreeRowData.path}" />
          <af:commandMenuItem actionListener=
            "#{dynamicContextMenuTable.alertTreeRowData}"
            text="Date -
              #{dynamicContextMenuTable.currentTreeRowData.lastModified}" />
        </af:menu>
      </af:menu>
    </af:popup>
  </f:facet>
  ...
</af:tree>
```

The code on the backing bean might look something like this:

```
public class DynamicContextMenuTableBean
{
  ...
  public void setCurrentTreeRowData(Map currentTreeRowData)
  {
    _currentTreeRowData = currentTreeRowData;
  }

  public Map getCurrentTreeRowData()
  {
    return _currentTreeRowData;
  }

  private Map _currentTreeRowData;
}
```

Tables and tree tables contain the `bodyContextMenu` facet. You can add a popup that contains a menu to this facet, and it will be displayed whenever a user clicks on the table, but not within a specific row.

For information about creating context menus, see [Declaratively Creating Popups](#).

Accessing Client Collection Components

With ADF Faces, the contents of collection-based components are rendered on the server. There may be cases when the client needs to access that content on the server, including:

- Client-side application logic may need to read the row-specific component state. For example, in response to row selection changes, the application may want to update the disabled or visible state of other components in the page (usually menu items or toolbar buttons). This logic may be dependent on row-specific metadata sent to the client using a stamped `inputHidden` component. In order to enable this, the application must be able to retrieve row-specific attribute values from stamped components.
- Client-side application logic may need to modify row-specific component state. For example, clicking a stamped command link in a table row may update the state of other components in the same row.
- The peer may need access to a component instance to implement event handling behavior (for information about peers, see [About Using ADF Faces Architecture](#)). For example, in order to deliver a client-side action event in response to a mouse click, the `AdfDhtmlCommandLinkPeer` class needs a reference to the component instance which will serve as the event source. The component also holds on to relevant state, including client listeners as well as attributes that control event delivery behavior, such as `disabled` or `partialSubmit`.

Because there is no client-side support for EL in the ADF Faces framework, nor is there support for sending entire table models to the client, the client-side code cannot rely on component stamping to access the value. Instead of reusing the same component instance on each row, a new JavaScript client component is created on each row (assuming any component must be created at all for any of the rows).

Therefore, to access row-specific data on the client, you need to use the stamped component itself to access the value. To do this without a client-side data model, you use a client-side selection change listener. For detailed instructions, see [Accessing Selected Values on the Client from Collection-Based Components](#).

Geometry Management for the Table, Tree, and Tree Table Components

By default, when tables, trees, and tree tables are placed in a component that stretches its children (for example, a `panelCollection` component inside a `panelStretchLayout` component), the table, tree, or tree table will stretch to fill the existing space. However, in order for the columns to stretch to fit the table, you must specify a specific column to stretch to fill up any unused space, using the `columnStretching` attribute. Otherwise, the table will only stretch vertically to fit as many rows as possible. It will not stretch the columns, as shown in [Figure 12-13](#).

Figure 12-13 Table Stretches But Columns Do Not

Name	Long String Name	Directory
.	. is a directory. It ...	true
..	.. is a directory. It...	true
admin.jar	admin.jar is a File. ...	false
applib	applib is a director...	true
applications	applications is a dir...	true
config	config is a director...	true
connectors	connectors is a dir...	true
database	database is a direc...	true
default-web-app	default-web-app is...	true
iiop.jar	iiop.jar is a File. It ...	false
iiop_gen_bin.jar	iiop_gen_bin.jar is ...	false
iiop_rmic.jar	iiop_rmic.jar is a Fil...	false
jazz	jazz is a directory. ...	true
jazz.jar	jazz.jar is a File. It...	false
jazzcore.jar	jazzcore.jar is a Fil...	false
jazzplugin.jar	jazzplugin.jar is a ...	false
jsp	jsp is a directory. I...	true
lib	lib is a directory. It...	true
loadbalancer.jar	loadbalancer.jar is...	false
log	log is a directory. I...	true
oc4j.jar	oc4j.jar is a File. It...	false
oc4jclient.jar	oc4jclient.jar is a F...	false
oc4j_interop.jar	oc4j_interop.jar is ...	false
ojspc.jar	ojspc.jar is a File. ...	false
persistence	persistence is a dir...	true
rmic.jar	rmic.jar is a File. It...	false
sql	sql is a directory. I...	true
.	. is a directory. It ...	true
..	.. is a directory. It...	true
admin.jar	admin.jar is a File. ...	false
applib	applib is a director...	true
applications	applications is a dir...	true
config	config is a director...	true
connectors	connectors is a dir...	true
database	database is a direc...	true

When placed in a component that does not stretch its children (for example, in a `panelCollection` component inside a `panelGroupLayout` component set to vertical), by default, a table width is set to 300px (27.27em units which translates to 300px for an 11px font setting) and the default fetch size is set to return 25 rows, as shown in [Figure 12-14](#).

Figure 12-14 Table Does Not Stretch

Name	Long String Name	Directory
.	. is a directory. It ...	true
..	.. is a directory. It ...	true
admin.jar	admin.jar is a File. ...	false
applib	applib is a director...	true
applications	applications is a dir...	true
config	config is a director...	true
connectors	connectors is a dir...	true
database	database is a direc...	true
default-web-app	default-web-app is...	true
iiop.jar	iiop.jar is a File. It ...	false
iiop_gen_bin.jar	iiop_gen_bin.jar is ...	false
iiop_rmic.jar	iiop_rmic.jar is a Fil...	false
jazn	jazn is a directory...	true
jazn.jar	jazn.jar is a File. It...	false
javax.jar	javax.jar is a File. It...	false

When you place a table in a component that does not stretch its children, you can control the height of the table so that is never more than a specified number of rows, using the `autoHeightRows` attribute. When you set this attribute to a positive integer, the table height will be determined by the number of rows set. If that number is higher than the `fetchSize` attribute, then only the number of rows in the `fetchSize` attribute will be returned. You can set `autoHeightRows` to `-1` (the default), to turn off auto-sizing.

Auto-sizing can be helpful in cases where you want to use the same table both in components that stretch their children and those that don't. For example, say you have a table that has 6 columns and can potentially display 12 rows. When you use it in a component that stretches its children, you want the table to stretch to fill the available space. If you want to use that table in a component that doesn't stretch its children, you want to be able to "fix" the height of the table. However, if you set a height on the table, then that table will not stretch when placed in the other component. To solve this issue, you can set the `autoHeightRows` attribute, which will be ignored when in a component that stretches, and will be honored in one that does not.

 **Note:**

The default value for the `autoHeightRows` attribute is handled by the `DEFAULT_DIMENSIONS` web.xml parameter. If you always want table components to be stretched when the parent can stretch, and to be the size of the `fetchSize` attribute when it cannot, set the `DEFAULT_DIMENSIONS` parameter instead of the `autoHeightRows` attribute. Set the `autoHeightRows` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `autoHeightRows` is `-1` (the table will not stretch). See [Geometry Management for Layout and Table Components](#).

On the other hand, when you do not want to set the `autoHeightRows` attribute but still want the table to stretch to fill up the empty vertical space, you can wrap the table with the `panelStretchLayout` component and use the `viewportBottom` attribute. However, you must explicitly use the `AFStretchWidth` styleClass and `dimensionsFrom` attributes with the `viewportBottom` attribute. When you use this setup, the distance from the bottom of the `panelStretchLayout` to the bottom of the viewport matches the number of the pixels in the `viewportBottom` attribute, so that the table resizes to consume that empty space.

This setting is however not recommended as a long-term solution because the user will see the adjustment that takes place when the table is resized after the table displays at an initial size.

The following example shows the `panelStretchLayout` component with `viewportBottom` attribute code.

```
<af:panelGroupLayout id="scroller" layout="scroll">
  <af:panelStretchLayout id="wrapper" styleClass="AFStretchWidth"
    <dimensionsFrom="parent" viewportBottom="64">
    <f:facet name="center">
      <af:table ...>...</af:table>
    </f:facet>
  </af:panelStretchLayout>
</af:panelGroupLayout>
```

Displaying Data in Tables

The ADF tables with collection-based components can be formatted using various visual aids. Column data can be stored and sorted. Only partial page rendering happens when the page containing tables is requested by a client.

The immediate children of a table component must be `column` components. Each visible column component is displayed as a separate column in the table. Column components contain components used to display content, images, or provide further functionality. For information about the features available with the column component, see [Columns and Column Data](#).

The child components of each column display the data for each row in that column. The column does not create child components per row; instead, the table uses stamping to render each row. Each child is stamped once per row, repeatedly for all the rows. As each row is stamped, the data for the current row is copied into a

property that can be addressed using an EL expression. You specify the name to use for this property using the `var` property on the table. Once the table has completed rendering, this property is removed or reverted back to its previous value.

Because of this stamping behavior, some components may not work inside the column. Most components will work without problems, for example any input and output components. If you need to use multiple components inside a cell, you can wrap them inside a `panelGroupLayout` component. Components that themselves support stamping are not supported, such as tables within a table. For information about using components whose values are determined dynamically at runtime, see [What You May Need to Know About Dynamically Determining Values for Selection Components in Tables](#).

You can use the `detailStamp` facet in a table to include data that can be optionally displayed or hidden. When you add a component to this facet, the table displays an additional column with an expand and collapse icon for each row. When the user clicks the icon to expand, the component added to the facet is displayed, as shown in [Figure 12-15](#).

Figure 12-15 Extra Data Can Be Optionally Displayed

Row Number	Size		Name	Parent Col5/6	
	Size In KB width some more text to make it wrap	Date Modified		Col5	Col6
0	0 B	7/12/2004	.	.	07/12/2004
1	0 B	7/12/2004	07/12/2004
2	1 KB	5/11/2004	admin.jar	admin.jar	05/11/2004
<div style="border: 1px solid #ccc; padding: 5px;"> <p>Name: <input type="text" value="admin.jar"/></p> <p>Size: <input type="text" value="1 KB"/></p> <p>Date Modified: <input type="text" value="5/11/2004"/></p> <p>Created by: <input type="text"/></p> </div>					
3	0 B	7/12/2004	applib	applib	07/12/2004

When the user clicks on the expanded icon to collapse it, the component is hidden, as shown in [Figure 12-16](#).

Figure 12-16 Extra Data Can Be Hidden

Row No.	Size		Name	Parent
	Size In KB width some more text to make it wrap	Date Modified		Col5
0	0 B	7/12/2004	.	.
1	0 B	7/12/2004
2	1 KB	5/11/2004	admin.jar	admin.jar
3	0 B	7/12/2004	applib	applib

For information about using the `detailStamp` facet, see [Adding Hidden Capabilities to a Table](#).

Columns and Column Data

Columns contain the components used to display the data. As stated previously, only one child component is needed for each item to be displayed; the values are stamped

as the table renders. Columns can be sorted, and you can configure whether the sort should be case-sensitive or case-insensitive. By default, it is case-sensitive.

Columns can also contain a filtering element. Users can enter a value into the filter and the returned data set will match the value entered in the filter. You can set the filter to be either case-sensitive or case-insensitive. If the table is configured to allow it, users can also reorder columns.

Columns have both header and footer facets. The header facet can be used instead of using the header text attribute of the column, allowing you to use a component that can be styled. The footer facet is displayed at the bottom of the column. For example, [Figure 12-17](#) uses footer facets to display the total at the bottom of two columns. If the number of rows returned is more than can be displayed, the footer facet is still displayed; the user can scroll to the bottom row.

Figure 12-17 Footer Facets in a Column

Name	ID1	ID2	Costs	Sales
name0	0	1	\$25,904.92	\$72,547.97
name1	1	11	\$20,762.95	\$81,852.96
name2	2	21	\$11,795.32	\$17,954.32
name3	3	31	\$40,374.90	\$34,145.20
Subtotal			\$98,838.09	\$206,500.46
name5	5	51	\$24,415.58	\$66,602.45
name6	6	61	\$17,848.06	\$84,103.58
name7	7	71	\$36,033.19	\$45,306.54
name8	8	81	\$25,880.95	\$45,904.55
Subtotal			\$104,177.79	\$241,917.12
name10	10	101	\$36,716.52	\$7,057.41
name11	11	111	\$9,196.06	\$78,574.38
name12	12	121	\$12,841.26	\$39,073.41
name13	13	131	\$21,831.43	\$18,521.42
Subtotal			\$80,585.27	\$143,226.61
name15	15	151	\$5,646.06	\$32,986.12
name16	16	161	\$8,849.79	\$57,981.35
name17	17	171	\$48,898.86	\$15,405.79
name18	18	181	\$44,246.57	\$63,763.24
Subtotal			\$107,641.27	\$170,136.49
name20	20	201	\$7,440.74	\$55,178.40
name21	21	211	\$10,324.60	\$12,813.21
name22	22	221	\$25,141.40	\$34,837.29
name23	23	231	\$16,351.75	\$6,366.33
Subtotal			\$59,258.50	\$109,195.23
Total			Total \$963,747.97	Total \$1,855,520.75

Formatting Tables

A table component offers many formatting and visual aids to the user. You can enable these features and specify how they can be displayed. These features include:

- **Row selection:** By default, at runtime, users cannot select rows. If you want users to be able to select rows in order to perform some action on them somewhere else on the page, or on another page, then enable row selection for the table by setting the `rowSelection` attribute. You can configure the table to allow either a single row or multiple rows to be selected. For information about how to then programmatically perform some action on the selected rows, see [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).
- **Scrolling/Pagination:** By default, on desktop devices, tables render a scroll bar that allows the user to scroll through all rows. On tablet devices, instead of a scroll bar, the table is paginated and displays a footer that allows the user to jump to specific pages of rows. You can change the default by setting the `scrollPolicy` attribute.

- **Table height:** You can set the table height to be absolute (for example, 300 pixels), or you can determine the height of the table based on the number of rows you wish to display at a time by setting the `autoHeightRows` attribute. See [Geometry Management for the Table, Tree, and Tree Table Components](#).

 **Note:**

When table is placed in a layout-managing container, such as a `panelSplitter` component, it will be sized by the container and the `autoHeightRows` is not honored.

 **Note:**

You cannot use JavaScript to dynamically size a table. The height of a table is set the first time is rendered and cannot be changed using JavaScript APIs.

- **Grid lines:** By default, an ADF table component draws both horizontal and vertical grid lines. These may be independently turned off using the `horizontalGridVisible` and `verticalGridVisible` attributes.
- **Banding:** Groups of rows or columns are displayed with alternating background colors using the `columnBandingInterval` attribute. This helps to differentiate between adjacent groups of rows or columns. By default, banding is turned off.
- **Column groups:** Columns in a table can be grouped into column groups, by nesting column components. Each group can have its own column group heading, linking all the columns together. See Step 14 in [How to Display a Table on a Page](#).
- **Editable cells:** When you elect to use input text components to display data in a table, you can configure the table so that all cells can be edited, or so that the user must explicitly click in the cell in order to edit it. See [Editing Data in Tables, Trees, and Tree Tables](#).

 **Performance Tip:**

When you choose to have cells be available for editing only when the user clicks on them, the table will initially load faster. This may be desirable if you expect the table to display large amounts of data.

- **Column stretching:** If the widths of the columns do not together fill the whole table, you can set the `columnStretching` attribute to determine whether or not to stretch columns to fill up the space, and if so, which columns should stretch. You can set the minimum width for columns, so that when there are many columns in a table and you enable stretching, columns will not be made smaller than the set minimum width. You can also set a width percentage for each column you want to stretch to determine the amount of space that column should take up when stretched.

 **Note:**

If the total sum of the columns' minimum widths equals more than the viewable space in the viewport, the table will expand outside the viewport and a scrollbar will appear to allow access outside the viewport.

 **Performance Tip:**

Column stretching is turned off by default. Turning on this feature may have a performance impact on the client rendering time when used for complex tables (that is, tables with a large amount of data, or with nested columns, and so on).

 **Note:**

Columns configured to be row headers or configured to be frozen will not be stretched because doing so could easily leave the user unable to access the scrollable body of the table.

- **Column selection:** You can choose to allow users to be able to select columns of data. As with row selection, you can configure the table to allow single or multiple column selection. You can also use the `columnSelectionListener` to respond to the `ColumnSelectionEvent` that is invoked when a new column is selected by the user. This event reports which columns were just deselected and which columns were just selected.
- **Column reordering:** Users can reorder the columns at runtime by simply dragging and dropping the column headers. By default, column reordering is allowed, and is handled by a menu item in the `panelCollection` component. See [Displaying Table Menus, Toolbars, and Status Bars](#).
- **Column freezing:** You can configure the table so that columns can be frozen and so will not scroll out of view. Columns can be frozen on either the left or right side of the table. This is controlled by the `freezeDirection` attribute on the table. You choose the column to start the freeze using the `frozen` attribute on the column.

 **Performance Tip:**

Use of column freezing increases the complexity of tables and can have a negative performance impact.

Formatting Columns

Each column component also offers many formatting and visual aids to the user. You can enable these features and specify how they can be displayed. These features include:

- **Column sorting:** Columns can be configured so that the user can sort the contents by a given column, either in ascending or descending order using the `sortable` attribute. A sort icon on a column header lets the user know that the column can be sorted. The sort icon toggles depending on the sort order. Other columns that can be sorted displays the sort icon on the mouse hover. When the user clicks on the icon to sort a previously unsorted column, the column's content is sorted in ascending order. Subsequent clicks on the same header sort the content in the reverse order. For sortable columns, a space is reserved on the right of the column header for the sort icon.

By default, sorting is case-sensitive. That is, `abc` would be sorted before `ABC`. You can configure the column so that instead, `abc` would be sorted the same as `ABC`, using the `sortStrength` attribute.

In order for the table to be able to sort, the underlying data model must also support sorting. See [What You May Need to Know About Programmatically Enabling Sorting for Table Columns](#).

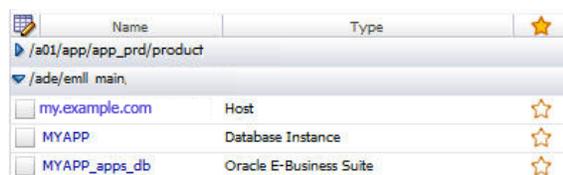
- **Content alignment:** You can align the content within the column to either the start, end, left, right, or center using the `align` attribute.

Tip:

Use `start` and `end` instead of `left` and `right` if your application supports multiple reading directions.

- **Column width:** The width of a column can be specified as an absolute value in pixels using the `width` attribute. If you configure a column to allow stretching, then you can also set the width as a percentage.
- **Column spanning:** You can configure a column to span across other columns using the `colSpan` attribute. Normally however, you use an EL expression as the value for the span, to enable only a certain cell in the column to actually span. For example, [Figure 12-18](#) shows a tree table whose `colSpan` value resolves to span all rows to the right, only if the node is a parent node.

Figure 12-18 Column Spans Only When the Node is a Parent



Name	Type	
/a01/app/app_prd/product		★
▼ /ade/emll main,		
<input type="checkbox"/> my.example.com	Host	★
<input type="checkbox"/> MYAPP	Database Instance	★
<input type="checkbox"/> MYAPP_apps_db	Oracle E-Business Suite	★

- **Line wrapping:** You can define whether or not the content in a column can wrap over lines, using the `noWrap` attribute. By default, content will not wrap.
- **Row headers:** You can define the left-most column to be a row header using the `rowHeader` attribute. When you do so, the left-most column is rendered with the same look as the column headers, and will not scroll off the page. [Figure 12-19](#) shows how a table showing departments appears if the first column is configured to be a row header.

Figure 12-19 Row Header in a Table

Dept. ID	Name	Manager	Location
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
150	Shareholder Services		1700

If you elect to use a row header column and you configure your table to allow row selection, the row header column displays a selection arrow when a users hovers over the row, as shown in [Figure 12-20](#).

Figure 12-20 Selection Icon in Row Header

Dept. ID	Name	Manager	Location
10	Administration	200	1700
20	Marketing	201	1800
→ 30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700

For tables that allow multiple selection, users can mouse down and then drag on the row header to select a contiguous blocks of rows. The table will also autoscroll vertically as the user drags up or down.

In addition, when an error occurs that results in a message for a component in the row, the icon for the severity is displayed in the row header. If more than one message exists, the icon for the maximum severity is displayed. [Figure 12-21](#) shows the error icon displayed in the row header because a date was entered incorrectly.

Figure 12-21 Row Headers Display Message Icons

Number	Name	Size	Date Modified	A Spinbox
 0	 .	0 B	Nov. 13	1979
1	 ..	0 B	7/12/2004	1979
2	 admin.jar	1 KB	5/11/2004	1979
3	 applib	0 B	7/12/2004	1979

 **Performance Tip:**

Use of row headers increases the complexity of tables and can have a negative performance impact.

 **Tip:**

While the user can change the way the table displays at runtime (for example the user can reorder columns or change column widths), those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).

How to Display a Table on a Page

You use the Create an ADF Faces Table dialog to add a table to a JSF page. You also use this dialog to add `column` components for each column you need for the table. You can also bind the table to the underlying model or bean using EL expressions.

 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create tables and the binding will be done for you. See [Creating ADF Databound Tables in *Developing Fusion Web Applications with Oracle Application Development Framework*](#).

Once you complete the dialog, and the table and columns are added to the page, you can use the Properties window to configure additional attributes of the table or columns, and add listeners to respond to table events. You must have an implementation of the `CollectionModel` class to which your table will be bound.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Data in Tables](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To display a table on a page:

1. Create a Java class that extends the `org.apache.myfaces.trinidad.model.CollectionModel` class.

Collection components use a `CollectionModel` class to access the data in the underlying collection. This class extends the JSF `DataModel` class, but is based on row keys instead of indexes to support underlying data changes. It also supports more advanced functionality, such as sorting.

You may also use other model classes, such as `java.util.List`, `array`, and `javax.faces.model.DataModel`. If you use one of these other classes, the collection component automatically converts the instance into a `CollectionModel` class, but without any additional functionality. For information about the `CollectionModel` class, see the MyFaces Trinidad javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

2. In the Components window, from the Data Views panel, drag and drop a **Table** onto the page.
3. Use the Create ADF Faces Table dialog to bind the table to any existing model you have. When you bind the table to a valid model, the dialog automatically shows the columns that will be created. You can then use the dialog to edit the values for the columns' `header` and `value` attributes, and choose the type of component that will be used to display the data. Alternatively, you can manually configure columns and bind at a later date. For help with the dialog, click **Help** or press F1.

 **Note:**

If you are using an `inputText` component to display a Character Large Object (CLOB), then you will need to create a custom converter that converts the CLOB to a `String`. For information about conversion, see [Creating Custom ADF Faces Converters](#).

4. In the Properties window, expand the Common section. If you have already bound your table to a model, the `value` attribute should be set. You can use this section to set the following table-specific attributes:
 - **RowSelection:** Set a value to make the rows selectable. Valid values are: `none`, `single`, and `multiple`, and `multipleNoSelectAll`.

 **Note:**

Users can select all rows and all columns in a table by clicking the column header for the row header if the `rowSelection` attribute is set to `multiple` and that table also contains a row header. If you do not want users to be able to select all columns and rows, then set `rowSelection` to `multipleNoSelectAll`.

For information about how to then programmatically perform some action on the selected rows, see [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).

- **ColumnSelection:** Set a value to make the columns selectable. Valid values are: `none`, `single`, and `multiple`.
5. Expand the Columns section. If you previously bound your table using the Create ADF Faces Table dialog, then these settings should be complete. You can use this section to change the binding for the table, to change the variable name used to access data for each row, and to change the display label and components used for each column.

 **Tip:**

If you want to use a component other than those listed, select any component in the Properties window, and then manually change it:

- a. In the Structure window, right-click the component created by the dialog and choose **Convert**.
- b. Select the desired component from the list. You can then use the Properties window to configure the new component.

 **Tip:**

If you want more than one component to be displayed in a column, add the other component manually and then wrap them both in a `panelGroupLayout` component. To do so:

- a. In the Structure window, right-click the first component and choose **Insert before** or **Insert after**. Select the component to insert.
- b. By default the components will be displayed vertically. To have multiple components displayed next to each other in one column, press the shift key and select both components in the Structure window. Right-click the selection and choose **Surround With**.
- c. Select `panelGroupLayout`.

6. Expand the Appearance section. You use this section to set the appearance of the table, by setting the following table-specific attributes:
 - **Width:** Specify the width of the table. You can specify the width as either a number of pixels or as a percentage. The default setting is 300 pixels. If you configure the table to stretch columns (using the `columnStretching` attribute), you must set the width to percentages.

 **Tip:**

If the table is a child to a component that stretches its children, then this width setting will be overridden and the table will automatically stretch to fit its container. For information about how components stretch, see [Geometry Management for the Table, Tree, and Tree Table Components](#).

- **ColumnStretching:** If the widths of the columns do not together fill the whole table, you can set this attribute to determine whether or not to stretch columns to fill up the space, and if so, which columns should stretch.

 **Note:**

If the table is placed inside a component that can stretch its children, only the table will stretch automatically. You must manually configure column stretching if you want the columns to stretch to fill the table.

 **Note:**

Columns configured to be row headers or configured to be frozen will not be stretched because doing so could easily leave the user unable to access the scrollable body of the table.

 **Performance Tip:**

Column stretching is turned off by default. Turning on this feature may have a performance impact on the client rendering time for complex tables.

You can set column stretching to one of the following values:

- `blank`: If you want to have an empty blank column automatically inserted and have it stretch (so the row background colors will span the entire width of the table).
- A specifically named column: Any column currently in the table can be selected to be the column to stretch.
- `last`: If you want the last column to stretch to fill up any unused space inside of the window.
- `none`: The default option where nothing will be stretched. Use this for optimal performance.
- `multiple`: All columns that have a percentage value set for their `width` attribute will be stretched to that percent, once other columns have been rendered to their (nonstretched) width. The percentage values will be weighted with the total. For example, if you set the `width` attribute on three columns to 50%, each column will get 1/3 of the remaining space after all other columns have been rendered.

 **Tip:**

While the widths of columns can change at runtime, those width values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see [Allowing User Customization on JSF Pages](#).

- **HorizontalGridVisible:** Specify whether or not the horizontal grid lines are to be drawn.
- **VerticalGridVisible:** Specify whether or not the vertical grid lines are to be drawn.
- **RowBandingInterval:** Specify how many consecutive rows form a row group for the purposes of color banding. By default, this is set to 0, which displays all rows with the same background color. Set this to 1 if you want to alternate colors.
- **ColumnBandingInterval:** Specify the interval between which the column banding occurs. This value controls the display of the column banding in the table. For example, `columnBandingInterval=1` would display alternately banded columns in the table.
- **FilterVisible:** You can add a filter to the table so that it displays only those rows that match the entered filter criteria. If you configure the table to allow filtering, you can set the filter to be case-insensitive or case-sensitive. For information, see [Enabling Filtering in Tables](#).
- **ScrollPolicy:** By default, on desktop devices, tables render a scroll bar that allows the user to scroll through all rows. On tablet devices, instead of a scroll bar, tables are rendered to display the table as paginated.

Set the value to `auto` to keep this default behavior. Set the value to `page` to have the table always display the rows as sets of pages, with a navigation to those pages in the footer. Set the value to `scroll` for tablet devices to have the table always render a scroll bar and scroll with implicit high-water mark scrolling; this setting is particularly useful to address performance problems with virtualized scrolling of table data on tablets. You can specify the number of rows to cache to minimize database roundtrips when the user scrolls back by setting a value for the `maxClientRows` attribute.

Set the value to `scrollPrefetch` for large tables when there are enough rows left to scroll to ensure smooth scrolling. Prefetching happens in parallel when a user is scrolling. When you set the value of the **scrollPolicy** attribute to `scrollPrefetch`, a table ensures the following:

- When a table is rendered initially, it fetches more rows than that is enough to fill the table viewport depending on the configuration of the table. For example, if the `fetchSize` is 25 and the viewport is 12 rows high, there are 13 rows left to scroll, which is more than half of `fetchSize` and no additional row beyond the first block is fetched. However, if the `fetchSize` is 25 and the viewport is 13 rows high, there are only 12 rows left to scroll, which is less than half of `fetchSize` and therefore, an additional row will be fetched.
- On subsequent scrolling of the table, the number of client rows left to scroll will be continually monitored and fetching of new data blocks is triggered when the number of rows left to scroll falls below the threshold. However, on tables with known row count, where virtualized scrollbar is supported, fetching will not be initiated until the user releases the mouse button from scrolling.

 **Note:**

For desktop devices, in order to explicitly set a table to display as paginated (configured as the default for tablet devices), you must set the `scrollPolicy` attribute to `page` and the `autoHeightRows` attribute to `0`. If these conditions are not met, the table will display with a scroll bar (whether it is a child to a stretched or a flowing component). For information about container components and tables, see [Geometry Management for the Table, Tree, and Tree Table Components](#).

- Text attributes: You can define text strings that will determine the text displayed when no rows can be displayed, as well as a table summary and description for accessibility purposes.
7. Expand the Behavior section. You use this section to configure the behavior of the table by setting the following table-specific attributes:
- **ColumnResizing:** Specify whether or not you want the end user to be able to resize a column's width at runtime. When set to `disabled`, the widths of the columns will be set once the page is rendered, and the user will not be able to change those widths.

 **Tip:**

While the user can change the values of the column width at runtime when `columnResizing` is set to `true`, those width values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see [Allowing User Customization on JSF Pages](#).

- **DisableColumnReordering:** By default, columns can be reordered at runtime using a menu option contained by default in the `panelCollection` component. You can change this so that users will not be able to change the order of columns. (The `panelCollection` component provides default menus and toolbar buttons for tables, trees, and tree tables. See [Displaying Table Menus, Toolbars, and Status Bars](#).)

 **Note:**

While the user can change the order of columns, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. See [Allowing User Customization on JSF Pages](#).

- **FetchSize:** Set the size of the block that should be returned with each data fetch. The default is 25.

 **Tip:**

You should determine the value of the `fetchSize` attribute by taking the height of the table and dividing it by the height of each row to determine how many rows will be needed to fill the table. If the `fetchSize` attribute is set too low, it will require multiple trips to the server to fill the table. If it is set too high, the server will need to fetch more rows from the data source than needed, thereby increasing time and memory usage. On the client side, it will take longer to process those rows and attach them to the component. See [Content Delivery](#). For slower database access set higher `fetchSize` in order to achieve continuous display and scrolling of data.

- **ContentDelivery:** Specify when the data should be delivered. When the `contentDelivery` attribute is set to `immediate`, data is fetched at the same time the component is rendered. If the `contentDelivery` attribute is set to `lazy`, data will be fetched and delivered to the client during a subsequent request. If the attribute is set to `whenAvailable` (the default), the renderer checks if the data is available. If it is, the content is delivered immediately. If it is not, then lazy delivery is used. See [Content Delivery](#).
- **AutoHeightRows:** Specify the number of rows to initially display in the table. When the returned number of rows exceeds this value, a scrollbar is displayed. If you want your table to size to be the same as the `fetchSize`, set it to 0. If you want the table to stretch to fill its parent container that is configured to stretch children, set it to -1 (for information about stretching the table, see [Geometry Management for the Table, Tree, and Tree Table Components](#)). Otherwise set it to a specific number that is lower than the current setting for `fetchSize`.

 **Note:**

Note the following about setting the `autoHeightRows` attribute:

- Specifying height on the `inlineStyle` attribute will have no effect and will be overridden by the value of `AutoHeightRows`.
- Specifying a `min-height` or `max-height` on the `inlineStyle` attribute is not recommended and is incompatible with the `autoHeightRows` attribute.
- When the component is placed in a layout-managing container, such as `panelSplitter`, it will be sized by the container (no auto-sizing will occur).

 **Note:**

The default value for the `autoHeightRows` attribute is handled by the `DEFAULT_DIMENSIONS` `web.xml` parameter. If you always want table components to be stretched when the parent can stretch, and to be the size of the `fetchSize` attribute when it cannot, set the `DEFAULT_DIMENSIONS` parameter to `auto`, instead of setting the `autoHeightRows` attribute.

When you set the `DEFAULT_DIMENSIONS` parameter to `auto` and place the table in a parent that does not stretch its children, and there is no override value for the `autoHeightRows` attribute, then the table will take its width from the `AFStretchWidth` style class, which by default, will stretch the width of the table to accommodate its child column components.

Set the `autoHeightRows` attribute when you want to override the global setting.

By default, `DEFAULT_DIMENSIONS` is set so that the value of `autoHeightRows` is `-1` (the table will not stretch). See [Geometry Management for Layout and Table Components](#).

- **DisplayRow:** Specify the row to be displayed in the table during the initial display. The possible values are `first` to display the first row at the top of the table, `last` to display the last row at the bottom of the table (users will need to scroll up to view preceding rows) and `selected` to display the first selected row in the table.

 **Note:**

The total number of rows from the table model must be known in order for this attribute to work successfully.

- **EditingMode:** Specify whether for any editable components, you want all the rows to be editable (`editAll`), you want the user to click a row to make it editable (`clickToEdit`), or you want the table to be rendered as read only (`readOnly`). For information, see [Editing Data in Tables, Trees, and Tree Tables](#).

 **Tip:**

If you choose `clickToEdit`, then only the active row can be edited. This row is determined by the `activeRowKey` attribute. By default, when the table is first rendered, the active row is the first visible row. When a user clicks another row, then that row becomes the active row. You can change this behavior by setting a different value for the `activeRowKey` attribute.

- **ContextMenuSelect:** Specify whether or not the row is selected when you right-click to open a context menu. When set to `true`, the row is selected. For

information about context menus, see [Using Popup Dialogs, Menus, and Windows](#).

- **FilterModel:** Use in conjunction with `filterVisible`. See [Enabling Filtering in Tables](#).
 - Various listeners: Bind listeners to methods that will execute when the table invokes the corresponding event. See [Handling Events](#).
8. Expand the Advanced section and set the following table-specific attributes:
- **ActiveRowKey:** If you choose `clickToEdit`, then only the active row can be edited. This row is determined by the `activeRowKey` attribute. By default, when the table is first rendered, the active row is the first visible row. When a user clicks another row, then that row becomes the active row. You can change this behavior by setting a different value for the `activeRowKey` attribute.
 - **DisplayRowKey:** Specify the row key to display in the table during initial display. This attribute should be set programmatically rather than declaratively because the value may not be strings. Specifying this attribute will override the `displayRow` attribute.

 **Note:**

The total number of rows must be known from the table model in order for this attribute to work successfully.

9. Expand the Other section and set the following:
- **BlockRowNavigationOnError:** Specify if you want users to be able to navigate away from a row that contains a validation error. When set to `always`, whenever a validation error occurs for a row, the user will always be blocked from navigating to a different row. When set to `never`, the user will never be blocked from navigating to a different row. When set to `auto` (the default), the framework will determine if the user can navigate.

For example, there may be cases when the table shares its values with another component on the page. You might have a table that allows the user to view a number of different records. When a specific record is selected, its information is displayed in a form. If the user changes some data in the form that causes an error, you do not want the user to then be able to scroll away from that record using the table. So for this example, you might set `BlockRowNavigationOnError` to `always`.
 - **FreezeDirection:** If you want columns to be able to be frozen, specify whether they should be frozen from the start of the table (the left side in a LTR locale) or the end of the table (the right side in a LTR locale). You must configure the column to start to the freeze using that column's `frozen` attribute.

For example, say you want the first three columns to be frozen. On the table, you would set `freezeDirection` to `start`, and on the third column, you would set `frozen` to `true`.

If you want the last four columns to be frozen, you would set `freezeDirection` to `end`, and on the fourth from last column, you would set `frozen` to `true`.
 - **SelectionEventDelay:** Set to `true` if you expect users to navigate through the table using the up and down arrow keys.

Users can navigate through the table using a mouse and the scrollbar, or using the up and down arrow keys. By default, a selection event is immediately fired when the user clicks a row. If the user is navigating through the rows using the arrow keys, this means that a selection event will be fired for each row, as the user navigates.

If you expect users to navigate through the table using the keys, you can set the `selectionEventDelay` attribute to `true`, so that there is a 300 millisecond delay before the selection event is fired. If the user navigates to another row within the 300 milliseconds, the selection event is canceled.

10. In the Structure window, select a column. In the Properties window, expand the Common section, and set the following column-specific attributes:
 - **HeaderText:** Specify text to be displayed in the header of the column. This is a convenience that generates output equivalent to adding a header facet containing an `outputText` component. If you want to use a component other than `outputText`, you should use the column's `header` facet instead (See Step 16). When the `header` facet is added, any value for the `headerText` attribute will not be rendered in a column header.
 - **Align:** Specify the alignment for this column. `start`, `end`, and `center` are used for left-justified, right-justified, and center-justified respectively in left-to-right display. The values `left` or `right` can be used when left-justified or right-justified cells are needed, irrespective of the left-to-right or right-to-left display. The default value is `null`, which implies that it is skin-dependent and may vary for the row header column versus the data in the column. For information about skins, see [Customizing the Appearance Using Styles and Skins](#).
 - **Sortable:** Specify whether or not the column can be sorted. A column that can be sorted has a header that when clicked, sorts the table by that column's property. Note that in order for a column to be sortable, the `sortable` attribute must be set to `true` and the underlying model must support sorting by this column's property. See [What You May Need to Know About Programmatically Enabling Sorting for Table Columns](#).

 **Note:**

When column selection is enabled, clicking on a column header selects the column instead of sorting the column. In this case, columns can be sorted by clicking the ascending/descending sort indicator.

- **SortStrength:** Specify the level of difference to be considered significant when sorting. Choose from one of the following (these values are the same as the values for the Java `Collator` object):
 - **Primary:** The sorting considers only the letter itself. Case and any accents are ignored: `abc`, `ÁBC`, `ábc`, and `ABC` will be sorted as `abc`, `ÁBC`, `ábc`, `ABC` (the order in which they appear). Use this for case-insensitive sorting.
 - **Secondary:** The sorting considers the letter and then any accent. Case is ignored: `abc`, `ÁBC`, `ábc`, and `ABC` will be sorted as `abc`, `ABC`, `ÁBC`, `ábc`. In locales that do not have accents, this will result in a case-insensitive search.

- **Tertiary:** The sorting will consider the letter, then the accent, and then the case: `abc`, `ÁBC`, `ábc`, and `ABC` will be sorted as `abc`, `ABC`, `ábc`, `ÁBC`. In locales that do not have accents, this will result in a case-sensitive search.
- **Identical:** The letters, accents, cases, and any other differences (such as words with punctuation) will be considered: `abc`, `ab-c`, `ÁBC`, `ábc`, and `ABC` will be sorted as `abc`, `ABC`, `ábc`, `ÁBC`, `ab-c`. This will result in a case-sensitive search, and is the default.
- **Filterable:** Specify whether or not the column can be filtered. A column that can be filtered has a filter field on the top of the column header. Note that in order for a column to be filterable, this attribute must be set to `true` and the `filterModel` attribute must be set on the table. Only leaf columns can be filtered and the filter component is displayed only if the column header is present. This column's `sortProperty` attribute must be used as a key for the `filterProperty` attribute in the `filterModel` class.

 **Note:**

For a column with filtering turned on (`filterable=true`), you can specify the input component to be used as the filter criteria input field. To do so, add a filter facet to the column and add the input component. See [Enabling Filtering in Tables](#).

11. Expand the Appearance section. Use this section to set the appearance of the column, using the following column-specific attributes:
 - **DisplayIndex:** Specify the display order index of the column. Columns can be rearranged and they are displayed in the table based on the `displayIndex` attribute. Columns without a `displayIndex` attribute value are displayed at the end, in the order in which they appear in the data source. The `displayIndex` attribute is honored only for top-level columns, because it is not possible to rearrange a child column outside of the parent column.
 - **Width:** Specify the width of the column. If the table uses column stretching, then you must enter a percentage for the width.

In column stretching, column width percentages are treated as weights. For example, if all columns are given 50% widths, and there are more than three columns, each column will receive an equal amount of space, while still respecting the value set for the `minWidth` attribute.

Because the width as a percentage is a weight rather than an actual percentage of space, if column stretching is turned on in the table, and only one column is listed as being stretched by having a percentage width, that column will use up all remaining space in the table not specified by pixel widths in the rest of the columns.
 - **MinimumWidth:** Specify the minimum number of pixels for the column width. When a user attempts to resize the column, this minimum width will be enforced. Also, when a column is flexible, it will never be stretched to be a size smaller than this minimum width. If a pixel width is defined and if the minimum width is larger, the minimum width will become the smaller of the two values. By default, the minimum width is 10 pixels.
 - **ShowRequired:** Specify whether or not an asterisk should be displayed in the column header if data is required for the corresponding attribute.

- **HeaderNoWrap** and **NoWrap**: Specify whether or not you want content to wrap in the header and in the column.
- **RowHeader**: Set to `true` if you want this column to be a row header for the table.

 **Performance Tip:**

Use of row headers increases the complexity of tables and can have a negative performance impact.

12. Expand the Behavior section. Use this section to configure the behavior of the columns, using the following column-specific attributes:
- **SortProperty**: Specify the property that is to be displayed by this column. This is the property that the framework might use to sort the column's data.
 - **Frozen**: Specify whether the column is frozen; that is it can't be scrolled off the page. In the table, columns up to the frozen column are locked with the header, and not scrolled with the rest of the columns. The frozen attribute is honored only on the top-level column, because it is not possible to freeze a child column by itself without its parent being frozen.

 **Note:**

By default, columns are frozen from this column to the left. That is, this column and any column to the left of it, will not scroll. You can change this by setting the `freezeDirection` attribute on the table component to `end`. By default, it is set to `start`.

 **Performance Tip:**

Use of frozen columns increases the complexity of tables and can have a negative performance impact.

- **Selected**: When set to `true`, the column will be selected on initial rendering.
13. If you want this column to span over subsequent columns, expand the Other section and set **ColSpan**. You can set it to the number of columns you want it to span, or you can set it to `ALL` to span to the end of the table. If you don't want all cells in the column to span, you can use an EL expression that resolves to a specific cell or cells.

The following example shows how you might set **colSpan** in a tree table component where you want only the parent node to span across all columns.

```
<af:column id="c1" sortable="true" sortProperty="Dname"
  colspan="#{testBean.container ? 'ALL' : '1'}"
  headerText="DepartmentName">
  <af:outputText value="#{node.Dname}" id="ot2"/>
</af:column>
```

The following example shows the corresponding managed bean code.

```
public class TestBean
{
    public boolean isContainer()
    {
        return _treeTable.isContainer();
    }
}
```

14. To add a column to an existing table, in the Structure window, right-click the table and choose **Insert Inside Table > Column**. To create column groups, drag a column component and drop it as a child to the component that will be the header. Continue to add columns to create the group. Set the `headerText` attribute on the parent column.
15. To add facets to the table, right-click the table and choose **Facets - Table** and choose the type of facet you want to add. You can then add a component directly to the facet.

 **Tip:**

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

16. To add facets to a column, right-click the column and choose **Facets - Column**, and choose the type of facet you want to add. You can then add a component directly to the facet.

 **Tip:**

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

17. Add components as children to the columns to display your data.

The component's value should be bound to the variable value set on the table's `var` attribute and the attribute to be displayed. For example, the table in the File Explorer application uses `file` as the value for the `var` attribute, and the first column displays the name of the file for each row. Therefore, the value of the output component used to display the directory name is `#{file.name}`.

 **Tip:**

If an input component is the direct child of a column, be sure its width is set to a width that is appropriate for the width of the column. If the width is set too large for its parent column, the browser may extend its text input cursor too wide and cover adjacent columns. For example, if an `inputText` component has its size set to 80 pixels and its parent column size is set to 20 pixels, the table may have an input cursor that covers the clickable areas of its neighbor columns.

To allow the input component to be automatically sized when it is not the direct child of a column, set `contentType="width:auto"`.

What Happens When You Add a Table to a Page

When you use JDeveloper to add a table onto a page, JDeveloper creates a table with a column for each attribute. If you bind the table to a model, the columns will reflect the attributes in the model. If you are not yet binding to model, JDeveloper will create the columns using the default values. You can change the default values (add/delete columns, change column headings, and so on) during in the table creation dialog or later using the Properties window.

The following example shows abbreviated page code for the table in the File Explorer application.

```
<af:table id="folderTable" var="file"
    value="#{explorer.contentViewManager.
        tableContentView.contentModel}"
    binding="#{explorer.contentViewManager.
        tableContentView.contentTable}"
    emptyText="#{explorerBundle['global.no_row']}"
    rowselection="multiple"
    contextMenuId=":context1" contentDelivery="immediate"
    columnStretching="last"
    selectionListener="#{explorer.contentViewManager.
        tableContentView.tableFileItem}"
    summary="table data">
<af:column width="180" sortable="true" sortStrength="identical"
    sortProperty="name"
    headerText="" align="start">
<f:facet name="header">
    <af:outputText value="#{explorerBundle['contents.name']}" />
</f:facet>
<af:panelGroupLayout>
    <af:image source="#{file.icon}"
        inlineStyle="margin-right:3px; vertical-align:middle;"
        shortDesc="file icon"/>
    <af:outputText value="#{file.name}" noWrap="true"/>
</af:panelGroupLayout>
</af:column>
<af:column width="70" sortable="true" sortStrength="identical"
    sortProperty="property.size">
<f:facet name="header">
    <af:outputText value="#{explorerBundle['contents.size']}" />
</f:facet>
    <af:outputText value="#{file.property.size}" noWrap="true"/>
</af:column>
```

```

...
<af:column width="100">
  <f:facet name="header">
    <af:outputText value="{explorerBundle['global.properties']}" />
  </f:facet>
  <af:link text="{explorerBundle['global.properties']}"
    partialSubmit="true"
    action="{explorer.launchProperties}"
    returnListener="{explorer.returnFromProperties}"
    windowWidth="300" windowHeight="300"
    useWindow="true">
  </af:link>
</af:column>
</af:table>

```

What Happens at Runtime: Data Delivery

When a page is requested that contains a table, and the content delivery is set to *lazy*, the page initially goes through the standard lifecycle. However, instead of fetching the data during that request, a special separate PPR request is run. Because the page has just rendered, only the Render Response phase executes, and the corresponding data is fetched and displayed. If the user's actions cause a subsequent data fetch (for example scrolling in a table), another PPR request is executed. [Figure 12-22](#) shows a page containing a table during the second PPR request. A message is displayed to let the user know that the data is being fetched.

Figure 12-22 Table Fetches Data in a Second PPR Request

Dept. ID	Name	Manager
Fetching Data...		

When the user clicks a sortable column header, the `table` component generates a `SortEvent` event. This event has a `getSortCriteria` property, which returns the criteria by which the table must be sorted, along with the sort strength. The table responds to this event by calling the `setSortCriteria()` method on the underlying `CollectionModel` instance, and calls any registered `SortListener` instances.

What You May Need to Know About Programmatically Enabling Sorting for Table Columns

Sorting can be enabled for a table column only if the underlying model supports sorting. If the model is a `CollectionModel` instance, it must implement the following methods:

- `public boolean isSortable(String propertyName)`

- `public List getSortCriteria()`
- `public void setSortCriteria(List criteria)`

The criteria in the second and third methods is a list where each item in the list is an instance of `org.apache.myfaces.trinidad.model.SortCriterion`, which supports sort strength.

See the MyFaces Trinidad website at <http://myfaces.apache.org/trinidad/index.html>.

If the model is not a `CollectionModel` instance, the table component wraps that model into an `org.apache.myfaces.trinidad.model.SortableModel` instance and converts the model to a `CollectionModel` instance that is sortable (`SortableModel` is a concrete class that extends `CollectionModel` and implements sorting functionality). In this case, the table will examine the actual data to determine which properties are sortable. Any column that has data that implements `java.lang.Comparable` will be sortable. This automatic support for sorting by the table is not as efficient as sorting directly into a `CollectionModel` instance but is sufficient for small data sets. Note that tables with a converted model allow sorting for only one column and therefore multi-column table sorting (normally done by supplying multiple sort criteria) is not supported on the converted model.

 **Note:**

When the underlying table model is not a `CollectionModel` instance and multi-column sorting is desired, consider using the table inside a `panelCollection` component. The panel user interface allows the user to sort using multiple sort criteria even though automatic sorting provided by the table with a converted model does not support it. For details about the `panelCollection` component, see [How to Add a panelCollection with a Table, Tree, or Tree Table](#).

What You May Need to Know About Performing an Action on Selected Rows in Tables

A collection-based component can allow users to select one or more rows and perform some actions on those rows (the carousel component does not support multiple selection).

When the selection state of a component changes, the component triggers selection events. A `selectionEvent` event reports which rows were just deselected and which rows were just selected.

To listen for selection events on a component, you can register a listener on the component either using the `selectionListener` attribute or by adding a listener to the component using the `addselectionListener()` method. The listener can then access the selected rows and perform some actions on them.

The current selection, that is the selected row or rows, are the `RowKeySet` object, which you obtain by calling the `getSelectedRowKeys()` method for the component. To change a selection programmatically, you can do either of the following:

- Add `rowKey` objects to, or remove `rowKey` objects from, the `RowKeySet` object.
- Make a particular row current by calling the `setRowIndex()` or the `setRowKey()` method on the component. You can then either add that row to the selection, or remove it from the selection, by calling the `add()` or `remove()` method on the `RowKeySet` object.

The following example shows a portion of a table in which a user can select some rows then click the **Delete** button to delete those rows. Note that the actions listener is bound to the `performDelete` method on the `mybean` managed bean.

```
<af:table binding="#{mybean.table}" rowselection="multiple" ...>
  ...
</af:table>
<af:button text="Delete" actionListener="#{mybean.performDelete}"/>
```

The following example shows an actions method, `performDelete`, which iterates through all the selected rows and calls the `markForDeletion` method on each one.

```
public void performDelete(ActionEvent action)
{
    UIXTable table = getTable();
    Iterator selection = table.getSelectedRowKeys().iterator();
    Object oldKey = table.getRowKey();
    try
    {
        while(selection.hasNext())
        {
            Object rowKey = selection.next();
            table.setRowKey(rowKey);
            MyRowImpl row = (MyRowImpl) table.getRowData();
            //custom method exposed on an implementation of Row interface.
            row.markForDeletion();
        }
    }
    finally
    {
        // restore the old key:
        table.setRowKey(oldKey);
    }
}
```

Note:

When using `setRowKey` and `setRowIndex` on `table/tree/treeTable` to change the current record for operation, you must restore the old current record after the operation. Otherwise, failure to restore the old currency can cause application errors.

What You May Need to Know About Dynamically Determining Values for Selection Components in Tables

There may be a case when you want to use a `selectOne` component in a table, but you need each row to display different choices in a component. Therefore, you need to dynamically determine the list of items at runtime.

While you may think you should use a `forEach` component to stamp out the individual items, this will not work because `forEach` does not work with the `CollectionModel` instance. It also cannot be bound to EL expressions that use component-managed EL variables, as those used in the table. The `forEach` component performs its functions in the JSF tag execution step while the table performs in the following component encoding step. Therefore, the `forEach` component will execute before the table is ready and will not perform its iteration function.

In the case of a `selectOne` component, the direct child must be the `items` component. While you could bind the `items` component directly to the row variable (for example, `<f:items value="#{row.Items}"/>`), doing so would not allow any changes to the underlying model.

Instead, you should create a managed bean that creates a list of items, as shown in the following example.

```
public List<SelectItem> getItems()
{
    // Grab the list of items
    FacesContext context = FacesContext.getCurrentInstance();
    Object rowItemObj = context.getApplication().evaluateExpressionGet(
        context, "#{row.items}", Object.class);
    if (rowItemObj == null)
        return null;
    // Convert the model objects into items
    List<SomeModelObject> list = (List<SomeModelObject>) rowItemObj;
    List<SelectItem> items = new ArrayList<SelectItem>(list.size());
    for (SomeModelObject entry : list)
    {
        items.add(new SelectItem(entry.getValue(), entry.getLabel()));
    }
    // Return the items
    return items;
}
```

You can then access the list from the one component on the page:

```
<af:table var="row">
  <af:column>
    <af:selectOneChoice value="#{row.myValue}">
      <f:Items value="#{page_backing.Items}"/>
    </af:selectOneChoice>
  </af:column>
</af:table>
```

What You May Need to Know About Read Only Tables

A table can be set as read-only by setting the `editingMode` attribute to `readOnly` value. If the `editingMode` attribute is set to `clickToEdit` type, then the table will result in a regular `clickToEdit` behavior.

A table can be rendered as editable, read-only, or edit-upon-clicking a row or a node type by setting an appropriate value on the `editingMode` attribute. When the `editingMode` attribute is set to `readOnly` value, all the components of that table will be rendered as read-only, including input components. If `editingMode` is set to `clickToEdit` value, then that table will result in a regular `clickToEdit` behavior.

Setting the `web.xml` parameter

`oracle.adf.view.rich.table.clickToEdit.initialRender.readOnly` will force all

`clickToEdit` type tables in an application to render as read-only only when the tables are initially loaded. All tables using other `editingMode` settings will be unaffected. See [Rendering Tables Initially as Read Only](#).

The `var` status of the table is updated with a new variable `readOnly` (`varStatus.readOnly`). This will provide information about the current status of the row if it is read-only or not. Applications can use this variable along with other conditional operators to decide which component to render. Following is example code using `varStatus.readOnly` to render an input text for an editable row and output text for a read-only row.

- You can add `varStatus` attribute in your table as follows:

```
<af:table value="#{bindings.EmpView1.collectionModel}" var="row"
  rows="#{bindings.EmpView1.rangeSize}"
  emptyText="#{bindings.EmpView1.viewable ? 'No data to
display.' : 'Access Denied.'}"
  fetchSize="#{bindings.EmpView1.rangeSize}"
  rowBandingInterval="0" id="t1" inlineStyle="width:1000px;"
  editingMode="clickToEdit"
  varStatus="vars" binding="#{backingBeanScope.Backing.t1}"
  partialTriggers=":cb4 :cb5 :cb6 :cb7 deptno1Id"
  columnSelection="single">
```

- You can use `varStatus` attribute in your column definition as follows:

```
<af:column headerText="switch" id="c100" width="130px">
  <af:inputText value="#{row.bindings.Hiredate.inputValue}"
    required="#{bindings.EmpView1.hints.Hiredate.mandatory}"
    columns="#{bindings.EmpView1.hints.Hiredate.displayWidth}"
    maximumLength="#{bindings.EmpView1.hints.Hiredate.precision}"
    shortDesc="#{bindings.EmpView1.hints.Hiredate.tooltip}"
    id="it2" rendered="#{!vars.readOnly}">
    <af:validator binding="#{row.bindings.Hiredate.validator}"/>
    <af:convertDateTime
pattern="#{bindings.EmpView1.hints.Hiredate.format}"/>
  </af:inputText>
  <af:outputText value="outputtext #{row.Hiredate}" id="ot2"
    rendered="#{vars.readOnly}">
    <af:convertDateTime
pattern="#{bindings.EmpView1.hints.Hiredate.format}"/>
  </af:outputText>
</af:column>
```

Adding Hidden Capabilities to a Table

While displaying the content in an ADF table built with collection-based components you can hide or display some content using `detailStamp` facet. The content appears with a toggle icon, when user clicks on it the hidden text is displayed in a popup window.

You can use the `detailStamp` facet in a table to include data that can be displayed or hidden. When you add a component to this facet, the table displays an additional column with a toggle icon. When the user clicks the icon, the component added to the

facet is shown. When the user clicks on the toggle icon again, the component is hidden. Figure 12-23 shows the additional column that is displayed when content is added to the detailStamp facet.

Figure 12-23 Table with Unexpanded DetailStamp Facet

Row Number	Size		Name	Parent Col5/6	
	Size In KB with some more text to make it wrap	Date Modified		Col5	Col6
0 ▶	0 B	7/12/2004	Ⓛ .	.	07/12/2004
1	0 B	7/12/2004	Ⓛ	07/12/2004
2 ▶	1 KB	5/11/2004	📄 admin.jar	admin.jar	05/11/2004
3	0 B	7/12/2004	Ⓛ applib	applib	07/12/2004
4 ▶	0 B	7/12/2004	Ⓛ applications	applications	07/12/2004

Figure 12-24 shows the same table, but with the detailStamp facet expanded for the first row.

Figure 12-24 Expanded detailStamp Facet

Row Number	Size		Name	Parent Col5/6	
	Size In KB with some more text to make it wrap	Date Modified		Col5	Col6
0 ▶	0 B	7/12/2004	Ⓛ .	.	07/12/2004
1	0 B	7/12/2004	Ⓛ	07/12/2004
2 📄	1 KB	5/11/2004	📄 admin.jar	admin.jar	05/11/2004
<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 5px;"> Name <input type="text" value="admin.jar"/> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 5px;"> Size <input type="text" value="1 KB"/> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 5px;"> Date Modified <input type="text" value="5/11/2004"/> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 5px;"> Created by <input type="text"/> </div>					
3	0 B	7/12/2004	Ⓛ applib	applib	07/12/2004

You can use an EL expression for the rendered attribute on the facet to determine whether or not to display the toggle icon and show details. For example, say on a shopping cart page you want to use the detailStamp facet to display gift wrapping information. However, not all order items will have gift wrapping information, so you only want the toggle icon to display if the order item has the information to display. You could create a method on managed bean that determines if there is information to display, and then bind the rendered attribute to that method. Figure 12-25 shows the same table but with icons displayed only for the rows that have information to display.

Figure 12-25 Conditional detailStamp Facet

Row No.	Size		Name	Parent Col5/6	
	Size In KB with some more text to make it wrap	Date Modified		Col5	Col6
0 ▶	0 B	7/12/2004	Ⓛ .	.	07/12/2004
1	0 B	7/12/2004	Ⓛ	07/12/2004
2 ▶	1 KB	5/11/2004	📄 admin.jar	admin.jar	05/11/2004
3	0 B	7/12/2004	Ⓛ applib	applib	07/12/2004
4 ▶	0 B	7/12/2004	Ⓛ applications	applications	07/12/2004
5	0 B	7/12/2004	Ⓛ config	config	07/12/2004

 **Note:**

If you set the table to allow columns to freeze, the freeze will not work when you display the `detailStamp` facet. That is, a user cannot freeze a column while the details are being displayed.

How to Use the `detailStamp` Facet

To use the `detailStamp` facet, you insert a component that is bound to the data to be displayed or hidden into the facet.

Before you begin:

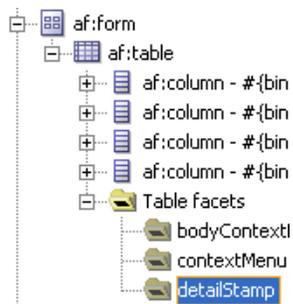
It may be helpful to have an understanding of how the attributes can affect functionality. See [Adding Hidden Capabilities to a Table](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To use the `detailStamp` facet:

1. In the Components window, drag the components you want to appear in the facet to the **detailStamp** facet in the Structure window, as shown in [Figure 12-26](#).

Figure 12-26 `detailStamp` Facet in the Structure Window

 **Note:**

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

 **Tip:**

If the facet does not appear in the Structure window, right-click the table and choose **Facets - Table > Detail Stamp**.

2. If the attribute to be displayed is specific to a current record, replace the JSF code (which simply binds the component to the attribute), so that it uses the table's variable to display the data for the current record.

The following example shows abbreviated code used to display the `detailStamp` facet shown in [Figure 12-24](#), which shows details about the selected row.

```
<af:table rowSelection="multiple" var="test1"
    value="#{tableTestData}"
  <f:facet name="detailStamp">
    <af:panelFormLayout rows="4" labelWidth="33%" fieldWidth="67%"
      inlineStyle="width:400px">
      <af:inputText label="Name" value="#{test1.name}"/>
      <af:group>
        <af:inputText label="Size" value="#{test1.size}"/>
        <af:inputText label="Date Modified" value="#{test1.inputDate}"/>
        <af:inputText label="Created by"/>
      </af:group>
    </af:panelFormLayout>
  </f:facet>
</af:table>
```

3. If you want the `detailStamp` facet to display its icon and components conditionally, set the `rendered` attribute on the facet to a method on a managed bean that will determine if the facet should be rendered.

 **Note:**

If your application uses the Fusion technology stack, then you can drag attributes from a data control and drop them into the `detailStamp` facet. You don't need to modify the code.

What Happens at Runtime: The `rowDisclosureEvent`

When the user hides or shows the details of a row, the table generates a `rowDisclosureEvent` event. The event tells the table to toggle the details (that is, either expand or collapse).

The `rowDisclosureEvent` event has an associated listener. You can bind the `rowDisclosureListener` attribute on the table to a method on a managed bean. This method will then be invoked in response to the `rowDisclosureEvent` event to execute any needed post-processing.

Enabling Filtering in Tables

An ADF table built with collection-based components can filter data in the table if the `filterModel` attribute object of the table is bound to an instance of the `FilterableQueryDescriptor` class.

You can add a filter to a table that can be used so that the table displays only rows whose values match the filter. When enabled and set to visible, a search criteria input field displays above each searchable column.

For example, the table in [Figure 12-27](#) has been filtered to display only rows in which the `Location` value is 1700.

Figure 12-27 Filtered Table

Dept. ID	Name	Manager	Location
10	Administration	200	1700
30	Purchasing	114	1700
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700

Filtered table searches are based on Query-by-Example and use the QBE text or date input field formats. The input validators are turned off to allow for entering characters for operators such as `>` and `<` to modify the search criteria. For example, you can enter `>1500` as the search criteria for a number column. Wildcard characters may also be supported. Searches can be either case-sensitive or case-insensitive. If a column does not support QBE, the search criteria input field will not render for that column.

The filtering feature uses a model for filtering data into the table. The table's `filterModel` attribute object must be bound to an instance of the `FilterableQueryDescriptor` class.

Note:

If your application uses the Fusion technology stack, then you can use data controls to create tables and filtering will be created for you. See [Creating ADF Databound Tables in *Developing Web User Interfaces with Oracle ADF Faces*](#)

In the following example, the table `filterVisible` attribute is set to `true` to enable the filter input fields. For each column to be filtered, you must set the `sortProperty` attribute to the associated column in the `filterModel` instance and the `filterable` attribute set to `true`.

```
<af:table value="#{myBean.products}" var="row"
  ...
  filterVisible="true"
  ...
  rowselection="single">
  ...
  <af:column sortProperty="ProductID" filterable="true" sortable="true"
    <af:outputText value="#{row.ProductID}" />
    ...
  </af:column>
  <af:column sortProperty="Name" filterable="true" sortable="true"
    <af:outputText value="#{row.Name}" />
    ...
  </af:column>
  <af:column sortProperty="warehouse" filterable="true" sortable="true"
    <af:outputText value="#{row.warehouse}" />
    ...
  </af:column>
</af:table>
```

How to Add Filtering to a Table

To add filtering to a table, first create a class that can provide the filtering functionality. You then bind the table to that class, and configure the table and columns to use filtering. The table that will use filtering must either have a value for its `headerText` attribute, or it must contain a component in the `header` facet of the column that is to be filtered. This allows the filter component to be displayed. Additionally, the column must be configured to be sortable, because the `filterModel` class uses the `sortProperty` attribute.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Enabling Filtering in Tables](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To add filtering to a table:

1. Create a Java class that is a subclass of the `FilterableQueryDescriptor` class.
The `ConjunctionCriterion` object returned from the `getFilterConjunctionCriterion` method must not be null. For information about this class, see the ADF Faces Javadoc.
2. Create a table, as described in [Displaying Data in Tables](#).
3. Select the table in the Structure window and set the following attributes in the Properties window:
 - **FilterVisible:** Set to `true` to display the filter criteria input field above searchable column.
 - **FilterModel:** Bind to an instance of the `FilterableQueryDescriptor` class created in Step 1.

 **Tip:**

If you want to use a component other than an `inputText` component for your filter (for example, an `inputDate` component), then instead of setting `filterVisible` to `true`, you can add the needed component to the `filter` facet. To do so:

- a. In the Structure window, right-click the column to be filtered and choose **Insert inside af:column > JSF Core > Filter facet**.
- b. From the Components window, drag and drop a component into the facet.
- c. Set the value of the component to the corresponding attribute within the `FilterableQueryDescriptor` class created in Step 1. Note that the value must take into account the variable used for the row, for example:

```
{af:inputDate label="Select Date" id="name"  
              value="row.filterCriteria.date"}
```

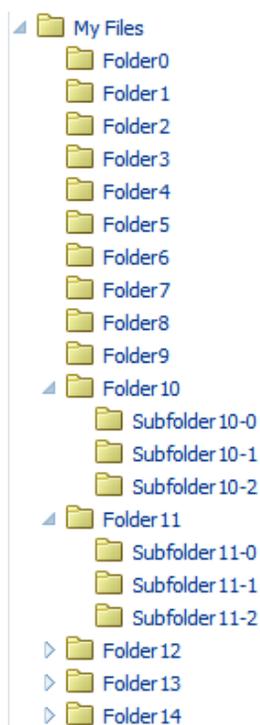
4. In the Structure window, select a column in the table and in the Properties window, and set the following for each column in the table:
 - **Filterable:** Set to `true`.
 - **FilterFeatures:** Set to `caseSensitive` or `caseInsensitive`. If not specified, the case sensitivity is determined by the model.

Displaying Data in Trees

Any hierarchical data can be displayed using ADF tree component.

The ADF Faces tree component displays hierarchical data, such as organization charts or hierarchical directory structures. In data of these types, there may be a series of top-level nodes, and each element in the structure may expand to contain other elements. For example, in an organization chart, any number of employees in the hierarchy may have any number of direct reports. The tree component can be used to show that hierarchy, where the direct reports appear as children to the node for the employee.

The tree component supports multiple root elements. It displays the data in a form that represents the structure, with each element indented to the appropriate level to indicate its level in the hierarchy, and connected to its parent. Users can expand and collapse portions of the hierarchy. [Figure 12-28](#) shows a tree used to display directories in the File Explorer application.

Figure 12-28 Tree Component in the File Explorer Application

The ADF Faces tree component uses a model to access the data in the underlying hierarchy. The specific model class is `oracle.adf.view.rich.model.TreeModel`, which extends `CollectionModel`, described in [Displaying Data in Tables](#).

You must create your own tree model to support your tree. The tree model is a collection of rows. It has an `isContainer()` method that returns `true` if the current row contains child rows. To access the children of the current row, you call the `enterContainer()` method. Calling this method results in the `TreeModel` instance changing to become a collection of the child rows. To revert back up to the parent collection, you call the `exitContainer()` method.

You may find the `org.apache.myfaces.trinidad.model.ChildPropertyTreeModel` class useful when constructing a `TreeModel` class, as shown in the following example.

```
List<TreeNode> root = new ArrayList<TreeNode>();
for(int i = 0; i < firstLevelSize; i++)
{
    List<TreeNode> level1 = new ArrayList<TreeNode>();
    for(int j = 0; j < i; j++)
    {
        List<TreeNode> level2 = new ArrayList<TreeNode>();
        for(int k=0; k<j; k++)
        {
            TreeNode z = new TreeNode(null, _nodeVal(i,j,k));
            level2.add(z);
        }
        TreeNode c = new TreeNode(level2, _nodeVal(i,j));
        level1.add(c);
    }
    TreeNode n = new TreeNode(level1, _nodeVal(i));
}
```

```
    root.add(n);
}
ChildPropertyTreeModel model = new ChildPropertyTreeModel(root, "children");
private String _nodeValue(Integer... args)
{
    StringBuilder s = new StringBuilder();
    for(Integer i : args)
        s.append(i);
    return s.toString();
}
```

 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create trees and the model will be created for you. See *Displaying Master-Detail Data in [Developing Web User Interfaces with Oracle ADF Faces](#)*

You can manipulate the tree similar to the way you can manipulate a table. You can do the following:

- To make a node current, call the `setRowIndex()` method on the tree with the appropriate index into the list. Alternatively, call the `setRowKey()` method with the appropriate `rowKey` object.
- To access a particular node, first make that node current, and then call the `getRowData()` method on the tree.
- To access rows for expanded or collapsed nodes, call `getAddedSet` and `getRemovedSet` methods on the `RowDisclosureEvent`. See [What You May Need to Know About Programmatically Expanding and Collapsing Nodes](#).
- To manipulate the node's child collection, call the `enterContainer()` method before calling the `setRowIndex()` and `setRowKey()` methods. Then call the `exitContainer()` method to return to the parent node.
- To point to a `rowKey` for a node inside the tree (at any level) use the `focusRowKey` attribute. The `focusRowKey` attribute is set when the user right-clicks on a node and selects the **Show as top** (or the **Show as top** toolbar button in the `panelCollection` component).

When the `focusRowKey` attribute is set, the tree renders the node pointed to by the `focusRowKey` attribute as the root node in the Tree and displays a Hierarchical Selector icon next to the root node. Clicking the Hierarchical Selector icon displays a Hierarchical Selector dialog which shows the path to the `focusRowKey` object from the root node of the tree. How this displays depends on the components placed in the `pathStamp` facet.

 **Note:**

You cannot use JavaScript to dynamically size a tree. The height of a tree is set the first time is rendered and cannot be changed using JavaScript APIs.

As with tables, trees use stamping to display content for the individual nodes. Trees contain a `nodeStamp` facet, which is a holder for the component used to display the data for each node. Each node is rendered (stamped) once, repeatedly for all nodes. As each node is stamped, the data for the current node is copied into a property that can be addressed using an EL expression. Specify the name to use for this property using the `var` property on the tree. Once the tree has completed rendering, this property is removed or reverted back to its previous value.

Because of this stamping behavior, only certain types of components are supported as children inside an ADF Faces tree. All components that have no behavior are supported, as are most components that implement the `ValueHolder` or `ActionSource` interfaces.

In the following example, the data for each element is referenced using the variable `node`, which identifies the data to be displayed in the tree. The `nodeStamp` facet displays the data for each element by getting further properties from the `node` variable:

```
<af:tree var="node">
  <f:facet name="nodeStamp">
    <af:outputText value="{node.firstname}"/>
  </f:facet>
</af:tree>
```

Trees also contain a `pathStamp` facet. This facet determines how the content of the Hierarchical Selector dialog is rendered, just like the `nodeStamp` facet determines how the content of the tree is rendered. The component inside the `pathStamp` facet can be a combination of simple `outputText`, `image`, and `outputFormatted` tags and cannot not be any input component (that is, any `EditableValueHolder` component) because no user input is allowed in the Hierarchical Selector popup. If this facet is not provided, then the Hierarchical Selector icon is not rendered.

For example, including an image and an `outputText` component in the `pathStamp` facet causes the tree to render an image and an `outputText` component for each node level in the Hierarchical Selector dialog. Use the same EL expression to access the value. For example, if you want to show the first name for each node in the path in an `outputText` component, the EL expression would be `<af:outputText value="{node.firstname}"/>`.

 **Tip:**

The `pathStamp` facet is also used to determine how default toolbar buttons provided by the `panelCollection` component will behave. If you want to use the buttons, add a component bound to a node value. For information about using the `panelCollection` component, see [Displaying Table Menus, Toolbars, and Status Bars](#).

How to Display Data in Trees

To create a tree, you add a tree component to your page and configure the display and behavior properties.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Data in Trees](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To add a tree to a page:

1. Create a Java class that extends the `org.apache.myfaces.trinidad.model.TreeModel` class.
2. In the Components window, from the Data Views panel, drag and drop a **Tree** to open the Insert Tree dialog.
3. Configure the tree as needed. For help with the dialog, click **Help** or press F1.
4. In the Properties window, expand the Data section and set the following attributes:
 - **Value:** Specify an EL expression for the object to which you want the tree to be bound. This must be an instance of `org.apache.myfaces.trinidad.model.TreeModel` as created in Step 1.
 - **Var:** Specify a variable name to represent each node.
 - **VarStatus:** Optionally enter a variable that can be used to determine the state of the component. During the Render Response phase, the tree iterates over the model rows and renders each node. For any given node, the `varStatus` attribute provides the following information:
 - `model`: A reference to the `CollectionModel` instance
 - `index`: The current row index
 - `rowKey`: The unique key for the current node
5. Expand the Appearance section and set the following attributes:
 - **DisplayRow:** Specify the node to display in the tree during the initial display. The possible values are `first` to display the first node, `last` to display the last node, and `selected` to display the first selected node in the tree. The default is `first`.
 - **DisplayRowKey:** Specify the row key to display in the tree during the initial display. This attribute should be set only programmatically. Specifying this attribute will override the `displayRow` attribute.
 - **Summary:** Optionally enter a summary of the data displayed by the tree.
6. Expand the Behavior section and set the following attributes:
 - **InitiallyExpanded:** Set to `true` if you want all nodes expanded when the component first renders.
 - **EditingMode:** Specify whether for any editable components used to display data in the tree, you want all the nodes to be editable (`editAll`), you want the user to click a node to make it editable (`clickToEdit`), or you want the tree to be rendered as read only (`readOnly`). See [Editing Data in Tables, Trees, and Tree Tables](#).
 - **ContextMenuSelect:** Determines whether or not the node is selected when you right-click to open a context menu. When set to `true`, the node is selected. For information about context menus, see [Using Popup Dialogs, Menus, and Windows](#).

- **RowSelection:** Set a value to make the nodes selectable. Valid values are: `none`, `single`, or `multiple`. For information about how to then programmatically perform some action on the selected nodes, see [What You May Need to Know About Programmatically Selecting Nodes](#).
- **ContentDelivery:** Specify when the data should be delivered. When the `contentDelivery` attribute is set to `immediate`, data is fetched at the same time the component is rendered. If the `contentDelivery` attribute is set to `lazy`, data will be fetched and delivered to the client during a subsequent request. If the attribute is set to `whenAvailable` (the default), the renderer checks if the data is available. If it is, the content is delivered immediately. If it is not, then lazy delivery is used. See [Content Delivery](#).
- **FetchSize:** Specify the number of rows in the data fetch block. See [Content Delivery](#).
- **AutoHeightRows:** Set to the maximum number of nodes to display before a scroll bar is displayed. The default value is `-1` (no automatic sizing for any number of number). You can set the value to `0` to have the value be the same as the `fetchSize` value.

 **Note:**

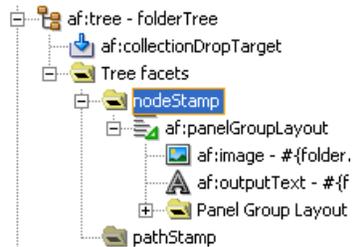
Note the following about setting the `autoHeightRows` attribute:

- Specifying height on the `inlineStyle` attribute will have no effect and will be overridden by the value of `AutoHeightRows`.
- Specifying a `min-height` or `max-height` on the `inlineStyle` attribute is incompatible with the `autoHeightRows` attribute., and should not be done.
- When the component is placed in a layout-managing container, such as `panelSplitter`, it will be sized by the container (no auto-sizing will occur). See [Geometry Management for the Table, Tree, and Tree Table Components](#).

- **SelectionListener:** Optionally enter an EL expression for a listener that handles selection events. See [What You May Need to Know About Programmatically Selecting Nodes](#).
 - **FocusListener:** Optionally enter an EL expression for a listener that handles focus events.
 - **RowDisclosureListener:** Optionally enter an EL expression for a listener method that handles node disclosure events.
7. Expand the Advanced section and set the following attributes:
- **FocusRowKey:** Optionally enter the node that is to be the initially focused node.
 - **DisclosedRowKeys:** Optionally enter an EL expression to a method on a backing bean that handles node disclosure. See [What You May Need to Know About Programmatically Expanding and Collapsing Nodes](#).
 - **SelectedRowKeys:** Optionally enter the keys for the nodes that should be initially selected. See [What You May Need to Know About Programmatically Selecting Nodes](#).

- To add components to display data in the tree, drag the desired component from the Components window to the **nodeStamp** facet. [Figure 12-29](#) shows the **nodeStamp** facet for the tree used to display directories in the File Explorer application.

Figure 12-29 nodeStamp Facet in the Structure Window



The component's value should be bound to the variable value set on the tree's `var` attribute and the attribute to be displayed. For example, the tree in the File Explorer application uses `folder` as the value for the `var` attribute, and displays the name of the directory for each node. Therefore, the value of the output component used to display the directory name is `#{folder.name}`.

 **Tip:**

Facets in a JSP or JSPX page can accept only one child component. Therefore, if you want to use more than one component per node, place the components in a group component that can be the facet's direct child, as shown in [Figure 12-29](#) (facets on a Facelets page can accept more than one child).

What Happens When You Add a Tree to a Page

When you add a tree to a page, JDeveloper adds a `nodeStamp` facet to stamp out the nodes of the tree. The following example shows the abbreviated code for the tree in the File Explorer application that displays the directory structure.

```
<af:tree id="folderTree"
  var="folder"
  binding="#{explorer.navigatorManager.foldersNavigator
    .foldersTreeComponent}"
  value="#{explorer.navigatorManager.foldersNavigator.
    foldersTreeModel}"
  disclosedRowKeys="#{explorer.navigatorManager.foldersNavigator.
    foldersTreeDisclosedRowKeys}"
  rowSelection="single"
  contextMenuId=":context2"
  selectionListener="#{explorer.navigatorManager.foldersNavigator.
    showSelectedFolderContent}">
  <f:facet name="nodeStamp">
    <af:panelGroupLayout>
      <af:image id="folderNodeStampImg" source="#{folder.icon}"
        inlineStyle="vertical-align:middle; margin-right:3px;
          shortDesc="folder icon"/>
    </af:panelGroupLayout>
  </f:facet>
</af:tree>
```

```

        <af:outputText id="folderNodeStampText" value="#{folder.name}"/>
    </af:panelGroupLayout>
</f:facet>
</af:tree>

```

What Happens at Runtime: Tree Component Events

The tree is displayed in a format with nodes indented to indicate their levels in the hierarchy. The user can click nodes to expand them to show children nodes. The user can click expanded nodes to collapse them. When a user clicks one of these icons, the component generates a `RowDisclosureEvent` event. You can register a custom `rowDisclosureListener` method to handle any processing in response to the event. See [What You May Need to Know About Programmatically Expanding and Collapsing Nodes](#).

When a user selects or deselects a node, the tree component invokes a `selectionEvent` event. You can register custom `selectionListener` instances, which can do post-processing on the tree component based on the selected nodes. See [What You May Need to Know About Programmatically Selecting Nodes](#).

What You May Need to Know About Programmatically Expanding and Collapsing Nodes

The `RowDisclosureEvent` event has two `RowKeySet` objects: the `RemovedSet` object for all the collapsed nodes and the `AddedSet` object for all the expanded nodes. The component expands the subtrees under all nodes in the added set and collapses the subtrees under all nodes in the removed set.

Your custom `rowDisclosureListener` method can do post-processing, on the tree component, as shown in the following example.

```

<af:treeTable id="folderTree" var="directory" value="#{fs.treeModel}"
    binding="#{editor.component}" rowselection="multiple"
    columnselection="multiple" focusRowKey="#{fs.defaultFocusRowKey}"
    selectionListener="#{fs.Table}"
    contextMenuId="treeTableMenu"
    rowDisclosureListener="#{fs.handleRowDisclosure}">

```

For the contraction of a tree node, you use `getRemovedSet` in a backing bean method that handles row disclosure events, as shown in the following example that illustrates the expansion of a tree node.

```

public void handleRowDisclosure(RowDisclosureEvent rowDisclosureEvent)
    throws Exception {
    Object rowKey = null;
    Object rowData = null;
    RichTree tree = (RichTree) rowDisclosureEvent.getSource();
    RowKeySet rks = rowDisclosureEvent.getAddedSet();

    if (rks != null) {
        int setSize = rks.size();
        if (setSize > 1) {
            throw new Exception("Unexpected multiple row disclosure
                added row sets found.");
        }
    }
}

```

```

    if (setSize == 0) {
        // nothing in getAddedSet indicates this is a node
        // contraction, not expansion. If interested only in handling
        // node expansion at this point, return.
        return;
    }

    rowKey = rks.iterator().next();
    tree.setRowKey(rowKey);
    rowData = tree.getRowData();

    // Do whatever is necessary for accessing tree node from
    // rowData, by casting it to an appropriate data structure
    // for example, a Java map or Java bean, and so forth.
}
}

```

Trees and tree tables use an instance of the `oracle.adf.view.rich.model.RowKeySet` class to keep track of which nodes are expanded. This instance is stored as the `disclosedRowKeys` attribute on the component. You can use this instance to control the expand or collapse state of a node in the hierarchy programmatically, as shown in the following example. Any node contained by the `RowKeySet` instance is expanded, and all other nodes are collapsed. The `addAll()` method adds all elements to the set, and the `removeAll()` method removes all the nodes from the set.

```

<af:tree var="node"
    inlineStyle="width:90%; height:300px"
    id="displayRowTable"
    varStatus="vs"
    rowselection="single"
    disclosedRowKeys="#{treeTableTestData.disclosedRowKeys}"
    value="#{treeTableTestData.treeModel}">

```

The backing bean method that handles the disclosed row keys is shown in the following example].

```

public RowKeySet getDisclosedRowKeys()
{
    if (disclosedRowKeys == null)
    {
        // Create the PathSet that we will use to store the initial
        // expansion state for the tree
        RowKeySet treeState = new RowKeySetTreeImpl();
        // RowKeySet requires access to the TreeModel for currency.
        TreeModel model = getTreeModel();
        treeState.setCollectionModel(model);
        // Make the model point at the root node
        int oldIndex = model.getRowIndex();
        model.setRowKey(null);
        for(int i = 1; i<=19; ++i)
        {
            model.setRowIndex(i);
            treeState.setContained(true);
        }
        model.setRowIndex(oldIndex);
        disclosedRowKeys = treeState;
    }
    return disclosedRowKeys;
}

```

What You May Need to Know About Programmatically Selecting Nodes

The tree and tree table components allow nodes to be selected, either a single node only, or multiple nodes. If the component allows multiple selections, users can select multiple nodes using Control+click and Shift+click operations.

When a user selects or deselects a node, the tree component fires a `selectionEvent` event. This event has two `RowKeySet` objects: the `RemovedSet` object for all the deselected nodes and the `AddedSet` object for all the selected nodes.

Tree and tree table components keep track of which nodes are selected using an instance of the class `oracle.adf.view.rich.model.RowKeySet`. This instance is stored as the `selectedRowKeys` attribute on the component. You can use this instance to control the selection state of a node in the hierarchy programmatically. Any node contained by the `RowKeySet` instance is deemed selected, and all other nodes are not selected. The `addAll()` method adds all nodes to the set, and the `removeAll()` method removes all the nodes from the set. Tree and tree table node selection works in the same way as table row selection. You can refer to sample code for table row selection in [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).

Displaying Data in Tree Tables

The ADF Tree Table component can display content in a hierarchical structure. Using this component you can display the content more elaborately than by using a tree component.

The ADF Faces tree table component displays hierarchical data in the form of a table. The display is more elaborate than the display of a tree component, because the tree table component can display columns of data for each tree node in the hierarchy. The component includes mechanisms for focusing on subtrees within the main tree, as well as expanding and collapsing nodes in the hierarchy. [Figure 12-30](#) shows the tree table used in the File Explorer application. Like the tree component, the tree table can display the hierarchical relationship between the files in the collection. And like the table component, it can also display attribute values for each file.

Figure 12-30 Tree Table in the File Explorer Application

Name	Type	Size (KB)	Date Modified
 File 11.doc	Document File	10	04/25/2013 5:57 PM
 File 11.js	JScript Script File	10	04/25/2013 5:57 PM
 Subfolder 11-0	File Folder		04/25/2013 5:57 PM
 File 11-0.jpg	Image File	100	04/25/2013 5:57 PM
 Subfolder 11-1	File Folder		04/25/2013 5:57 PM
 File 11-1.jpg	Image File	100	04/25/2013 5:57 PM
 Subfolder 11-2	File Folder		04/25/2013 5:57 PM

The immediate children of a tree table component must be column components, in the same way as for table components. Unlike the table, the tree table component has a `nodeStamp` facet which holds the column that contains the primary identifier of a node

in the hierarchy. The `treeTable` component supports the same stamping behavior as the `Tree` component (see [Displaying Data in Trees](#)).



Note:

The `nodeStamp` facet can only contain one column (which becomes the node in the tree).

For example, in the File Explorer application (as shown in [Figure 12-30](#)), the primary identifier is the file name. This column is what is contained in the `nodeStamp` facet. The other columns, such as **Type** and **Size**, display attribute values on the primary identifier, and these columns are the direct children of the tree table component. This tree table uses `node` as the value of the variable that will be used to stamp out the data for each node in the `nodeStamp` facet column and each component in the child columns. The following example shows abbreviated code for the tree table in the File Explorer application.

```
<af:treeTable id="folderTreeTable" var="file"
    value="#{explorer.contentViewManager.treeTableContentView.
        contentModel}"
    binding="#{explorer.contentViewManager.treeTableContentView.
        contentTreeTable}"
    emptyText="#{explorerBundle['global.no_row']}"
    columnStretching="last"
    rowSelection="single"
    selectionListener="#{explorer.contentViewManager.
        treeTableContentView.treeTableSelectFileItem}"
    summary="treeTable data">
  <f:facet name="nodeStamp">
    <af:column headerText="#{explorerBundle['contents.name']}"
        width="200" sortable="true" sortProperty="name">
      <af:panelGroupLayout>
        <af:image source="#{file.icon}"
            shortDesc="#{file.name}"
            inlineStyle="margin-right:3px; vertical-align:middle;"/>
        <af:outputText id="nameStamp" value="#{file.name}"/>
      </af:panelGroupLayout>
    </af:column>
  </f:facet>
  <f:facet name="pathStamp">
    <af:panelGroupLayout>
      <af:image source="#{file.icon}"
          shortDesc="#{file.name}"
          inlineStyle="margin-right:3px; vertical-align:middle;"/>
      <af:outputText value="#{file.name}"/>
    </af:panelGroupLayout>
  </f:facet>
  <af:column headerText="#{explorerBundle['contents.type']}">
    <af:outputText id="typeStamp" value="#{file.type}"/>
  </af:column>
  <af:column headerText="#{explorerBundle['contents.size']}">
    <af:outputText id="sizeStamp" value="#{file.property.size}"/>
  </af:column>
  <af:column headerText="#{explorerBundle['contents.lastmodified']}"
      width="140">
    <af:outputText id="modifiedStamp"
        value="#{file.property.lastModified}"/>
  </af:column>
</af:treeTable>
```

```
</af:column>  
</af:treeTable>
```

The tree table component supports many of the same attributes as both tables and trees. For additional information about these attributes see [Displaying Data in Tables](#) and [Displaying Data in Trees](#).

How to Display Data in a Tree Table

You use the Insert Tree Table wizard to create a tree table. Once the wizard is complete, you can use the Properties window to configure additional attributes on the tree table.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Data in Tree Tables](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To add a tree table to a page:

1. In the Components window, from the Data Views panel, drag and drop a **Tree Table** onto the page to open the Insert Tree Table wizard.
2. Configure the table by completing the wizard. For help with the wizard, click **Help** or press F1.
3. Use the Properties window to configure any other attributes.

Tip:

The attributes of the tree table are the same as those on the table and tree components. Refer to [How to Display a Table on a Page](#), and [How to Display Data in Trees](#) for help in configuring the attributes.

Passing a Row as a Value

The ADF allows you to pass the data of an entire row of a table or a tree residing on another page within the framework. You can achieve this by using the `setPropertyListener` tag. This is helpful when you want to retrieve the data in a table by a click of a button.

There may be a case where you need to pass an entire row from a collection as a value. To do this, you pass the variable used in the table to represent the row, or used in the tree to represent a node, and pass it as a value to a property in the `pageFlow` scope. Another page can then access that value from the scope. The `setPropertyListener` tag allows you to do this (for more information about the `setPropertyListener` tag, including procedures for using it, see [Passing Values Between Pages](#)).

For example, suppose you have a master page with a single-selection table showing employees, and you want users to be able to select a row and then click a button to navigate to a new page to edit the data for that row, as shown in the following

example. The EL variable name `emp` is used to represent one row (employee) in the table. The `action` attribute value of the `button` component is a static string `showEmpDetail`, which allows the user to navigate to the Employee Detail page. The `setPropertyListener` tag takes the `from` value (the variable `emp`), and stores it with the `to` value.

```
<af:table value="#{myManagedBean.allEmployees}" var="emp"
    rowSelection="single">
  <af:column headerText="Name">
    <af:outputText value="#{emp.name}"/>
  </af:column>
  <af:column headerText="Department Number">
    <af:outputText value="#{emp.deptno}"/>
  </af:column>
  <af:column headertext="Select">
    <af:button text="Show more details" action="showEmpDetail">
      <af:setPropertyListener from="#{emp}"
        to="#{pageFlowScope.empDetail}"
        type="action"/>
    </af:button>
  </af:column>
</af:table>
```

When the user clicks the button on an employee row, the listener executes, and the value of `#{emp}` is retrieved, which corresponds to the current row (employee) in the table. The retrieved row object is stored as the `empDetail` property of `pageFlowScope` with the `#{pageFlowScope.empDetail}` EL expression. Then the action event executes with the static outcome, and the user is navigated to a detail page. On the detail page, the `outputText` components get their value from `pageFlowScope.empDetail` objects, as shown in the following example.

```
<h:panelGrid columns="2">
  <af:outputText value="Firstname:"/>
  <af:inputText value="#{pageFlowScope.empDetail.name}"/>
  <af:outputText value="Email:"/>
  <af:inputText value="#{pageFlowScope.empDetail.email}"/>
  <af:outputText value="Hiredate:"/>
  <af:inputText value="#{pageFlowScope.empDetail.hiredate}"/>
  <af:outputText value="Salary:"/>
  <af:inputText value="#{pageFlowScope.empDetail.salary}"/>
</h:panelGrid>
```

Displaying Table Menus, Toolbars, and Status Bars

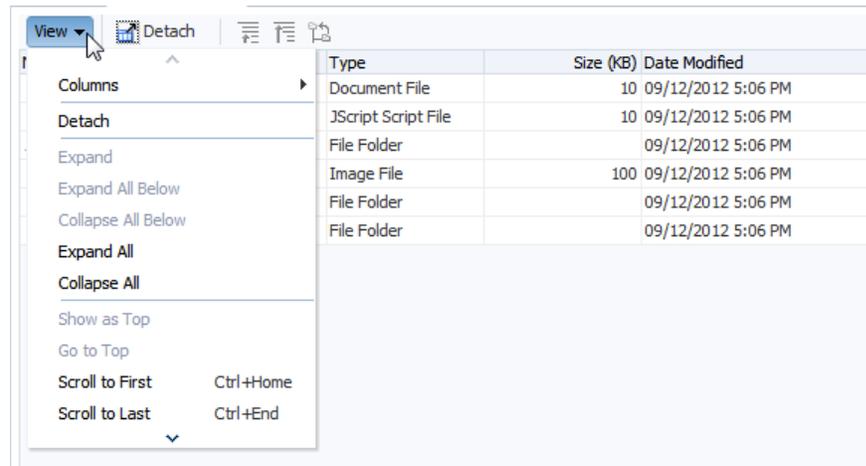
Using ADF `panelCollection` component you can add menus, toolbars, and status bars to a table. You can also add actions, such as expanding or collapsing the content, detaching the toolbar from the tree or tree table, and so on.

You can use the `panelCollection` component to add menus, toolbars, and status bars to tables, trees, and tree tables. To use the `panelCollection` component, you add the table, tree, or tree table component as a direct child of the `panelCollection` component. The `panelCollection` component provides default menus and toolbar buttons.

[Figure 12-31](#) shows the `panelCollection` component with the tree table component in the File Explorer application. The toolbar contains a menu that provides actions that can be performed on the tree table (such as expanding and collapsing nodes), a button that allows users to detach the tree table, and buttons that allow users to

change the rows displayed in the tree table. You can configure the toolbar to not display certain toolbar items. For example, you can turn off the buttons that allow the user to detach the tree or table. For information about menus, toolbars, and toolbar buttons, see [Using Menus, Toolbars, and Toolboxes](#).

Figure 12-31 Panel Collection for Tree Table with Menus and Toolbar



Among other facets, the `panelCollection` component contains a `menu` facet to hold menu components, a `toolbar` facet for toolbar components, a `secondaryToolbar` facet for another set of toolbar components, and a `statusbar` facet for status items.

The default top-level menu and toolbar items vary depending on the component used as the child of the `panelCollection` component:

- Table and tree: Default top-level menu is **View**.
- Table and tree table with selectable columns: Default top-level menu items are **View** and **Format**.
- Table and tree table: Default toolbar menu is **Detach**.
- Table and tree table with selectable columns: Default top-level toolbar items are **Freeze**, **Detach**, and **Wrap**.
- Tree and tree table (when the `pathStamp` facet is used): The toolbar buttons **Go Up**, **Go To Top**, and **Show as Top** also appear.

The following example shows how the `panelCollection` component contains menus and toolbars.

```
<af:panelCollection
  binding="#{editor.component}">
  <f:facet name="viewMenu">
    <af:group>
      <af:commandMenuItem text="View Item 1..." />
      <af:commandMenuItem text="View Item 2..." />
      <af:commandMenuItem text="View Item 3..." disabled="true" />
      <af:commandMenuItem text="View Item 4..." />
    </af:group>
  </f:facet>

  <f:facet name="menus">
```

```

<af:menu text="Actions">
  <af:commandMenuItem text="Add..." />
  <af:commandMenuItem text="Create.." />
  <af:commandMenuItem text="Update..." disabled="true"/>
  <af:commandMenuItem text="Copy"/>
  <af:commandMenuItem text="Delete"/>
  <af:commandMenuItem text="Remove" accelerator="control A"/>
  <af:commandMenuItem text="Preferences"/>
</af:menu>
</f:facet>
<f:facet name="toolbar">
  <af:toolbar>
    <af:button shortDesc="Create" icon="/new_ena.png">
    </af:button>
    <af:button shortDesc="Update" icon="/update_ena.png">
    </af:button>
    <af:button shortDesc="Delete" icon="/delete_ena.png">
    </af:button>
  </af:toolbar>
</f:facet>
<f:facet name="secondaryToolbar">
</f:facet>
<f:facet name="statusbar">
  <af:toolbar>
    <af:outputText id="statusText" ... value="Custom Statusbar Message"/>
  </af:toolbar>
</f:facet>
<af:table rowselection="multiple" columnselection="multiple"
          ...
<af:column
          ...
</af:column>

```

 **Tip:**

You can make menus detachable in the `panelCollection` component. See [Using Menus in a Menu Bar](#). Consider using detached menus when you expect users to do any of the following:

- Execute similar commands repeatedly on a page.
- Execute similar commands on different rows of data in a large table, tree table, or tree.
- View data in long and wide tables or tree tables, and trees. Users can choose which columns or branches to hide or display with a single click.
- Format data in long or wide tables, tree tables, or trees.

How to Add a `panelCollection` with a Table, Tree, or Tree Table

You add a `panelCollection` component and then add the table, tree, or tree table inside the `panelCollection` component. You can then add and modify the menus and toolbars for it.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Table Menus, Toolbars, and Status Bars](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To create a `panelCollection` component with an aggregate display component:

1. In the Components window, from the Layout panel, drag and drop a **Panel Collection** onto the page. Add the table, tree, or tree table as a child to that component.

Alternatively, if the table, tree, or tree table already exists on the page, you can right-click the component and choose **Surround With**. Then select **Panel Collection** to wrap the component with the `panelCollection` component.

2. Optionally, customize the `panelCollection` toolbar by turning off specific toolbar and menu items. To do so, select the `panelCollection` component in the Structure window. In the Properties window, set the `featuresOff` attribute. [Table 12-1](#) shows the valid values and the corresponding effect on the toolbar.

Table 12-1 Valid Values for the `featuresOff` Attribute

Value	Will not display...
<code>statusBar</code>	status bar
<code>viewMenu</code>	View menu
<code>formatMenu</code>	Format menu
<code>columnsMenuItem</code>	Columns menu item in the View menu
<code>columnsMenuItem:colId</code> For example: <code>columnsMenuItem:col1</code> , <code>col2</code>	Columns with matching IDs in the Columns menu For example, the value to the left would not display the columns whose IDs are <code>col1</code> and <code>col2</code>
<code>freezeMenuItem</code>	Freeze menu item in the View menu
<code>detachMenuItem</code>	Detach menu item in the View menu
<code>sortMenuItem</code>	Sort menu item in the View menu
<code>reorderColumnsMenuItem</code>	Reorder Columns menu item in the View menu
<code>resizeColumnsMenuItem</code>	Resize Columns menu item in the Format menu
<code>wrapMenuItem</code>	Wrap menu item in the Format menu
<code>showAsTopMenuItem</code>	Show As Top menu item in the tree's View menu
<code>scrollToFirstMenuItem</code>	Scroll To First menu item in the tree's View menu
<code>scrollToLastMenuItem</code>	Scroll To Last menu item in the tree's View menu
<code>freezeToolBarItem</code>	Freeze toolbar item
<code>detachToolBarItem</code>	Detach toolbar item
<code>wrapToolBarItem</code>	Wrap toolbar item
<code>showAsTopToolBarItem</code>	Show As Top toolbar item
<code>wrap</code>	Wrap menu and toolbar items
<code>freeze</code>	Freeze menu and toolbar items
<code>detach</code>	Detach menu and toolbar items

3. Add your custom menus and toolbars to the component:

- **Menus:** Add a `menu` component inside the `menu` facet.
- **Toolbars:** Add a `toolbar` component inside the `toolbar` or `secondaryToolbar` facet.
- **Status items:** Add items inside the `statusbar` facet.
- **View menu:** Add `commandMenuItem` components to the `viewMenu` facet. For multiple items, use the `group` component as a container for the `commandMenuItem` components.

From the Components window, drag and drop the component into the facet. For example, drop **Menu** into the `menu` facet, then drop **Menu Items** into the same facet to build a menu list. For instructions about menus and toolbars, see [Using Menus, Toolbars, and Toolboxes](#).

Displaying a Collection in a List

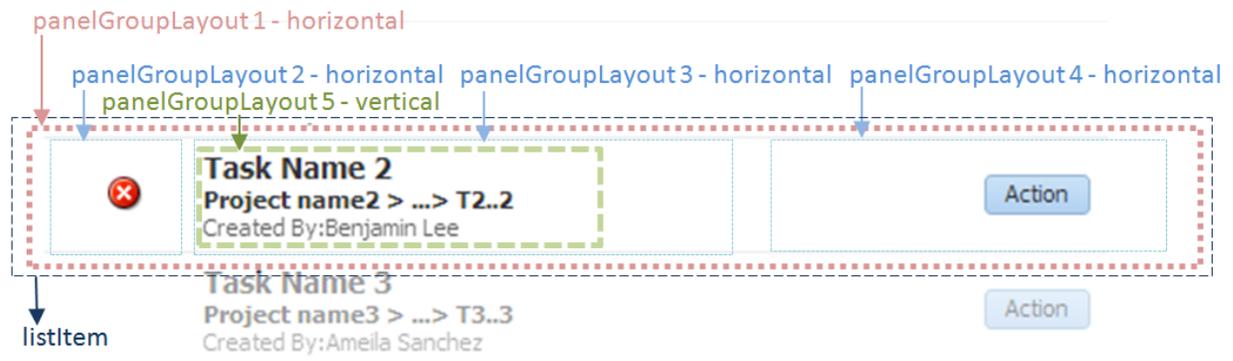
Using the ADF `listView` and `listItem` components you can display the contents in a single-column table structure. You can also achieve two-level hierarchy by binding `listView` to a `TreeModel`.

Instead of using a table with multiple columns, you can use the `listView` and `listItem` components to display structured data in a simple table-like format that contains just one column. [Figure 12-32](#) shows a `listView` component that contains one `listItem` component used to display an error icon, task information, and an action button, for each row.

Figure 12-32 The `listView` Component with a `listItem` Component

<p>Task Name 1 Project name1 > ...> T1..1 Created By: Annett Barnes</p>	Action
<p> Task Name 2 Project name2 > ...> T2..2 Created By: Benjamin Lee</p>	Action
<p>Task Name 3 Project name3 > ...> T3..3 Created By: Amelia Sanchez</p>	Action
<p>Task Name 4 Project name4 > ...> T4..4 Created By: Jacob Miller</p>	Action

As shown in [Figure 12-33](#), instead of using columns to group the data to be displayed, a mix of layout components and other components, held by one `listItem` component, display the actual the data. In this example, the `listItem` component contains one large `panelGroupLayout` component set to display its children horizontally. The children are three other `panelGroupLayout` components used to group their children as columns might in a table. These `panelGroupLayout` components are also set to display their children horizontally. The second of these layout components contains one `panelGroupLayout` component set to display its child components (in this case three `outputText` components) vertically.

Figure 12-33 The listItem Component Contains Multiple Components That Display the Data

See [Example 12-1](#) below, for the corresponding code.

You bind the `listView` component to the collection. The component then repeatedly renders one `listItem` component by stamping the value for each item. As each item is stamped, the data for the current row is copied into a property that can be addressed by an EL expression that uses the `listView` component's `var` attribute. Once the list has completed rendering, this property is removed or reverted back to its previous value.

In this example, the `listView` value is bound to the `demolistView.taskModel` object. The properties on this object can be accessed using the `var` property, which is set to `item`. For example, in order to display the task name, the `outputText` component value is set to `item.taskName`.

The `listView` component can also display a limited, two-level hierarchy. To display a hierarchy, the `listView` needs to be bound to a `TreeModel` instead of a `CollectionModel`. The `TreeModel` can contain one root level and one child level (for information about the `TreeModel` class, see [Displaying Data in Trees](#)).

As with trees, the `listView` uses stamping to display content for the individual nodes, and a facet (named the `groupHeaderStamp` facet) that acts as a holder for the component used to display the parent group for the nodes. However, since the `listView` only allows two levels, the `groupHeaderStamp` facet contains the component used to display only the root level.

[Figure 12-34](#) shows a `listView` component displaying a simple hierarchy that has letters of the alphabet as the root, and employee objects as the leaf nodes.

Figure 12-34 Simple Hierarchy in a listView Component

The components used to display the employee object are placed in a `listItem` component, while the components used to display the letter of the alphabet are placed in a `listItem` component inside the `groupHeaderStamp` facet, as shown in the following example.

```
<af:listView id="listView" binding="#{editor.component}"
    var="item" varStatus="vs" groupDisclosurePolicy="noDisclosure"
    value="#{demolistView.ABTreeModel}">
  <af:listItem id="listItem1">
    <af:panelGroupLayout id="pgl3" layout="vertical">
      <af:outputText id="ot2" value="#{item.ename}" styleClass="ABName"/>
      <af:outputText id="ot3" value="#{item.job}" styleClass="ABJob"/>
    </af:panelGroupLayout>
  </af:listItem>
  <f:facet name="groupHeaderStamp">
    <af:listItem id="listItem2" styleClass="ABHeader">
      <af:outputText id="ot1" value="#{item.alphabetHeading}"/>
    </af:listItem>
  </f:facet>
</af:listView>
```

When you display a hierarchy in a `listView` component, you can configure it so that the headers can disclose or hide its child components, as shown in [Figure 12-35](#).

Figure 12-35 The listView Component Configured to Provide Collapsing Headers

By default, the `listView` component is configured to display all children. You can change this using the `groupDisclosurePolicy` attribute.

When a user collapses or expands a group, a `RowDisclosureEvent` is fired. You can use the `groupDisclosureListener` to programmatically expand and collapse nodes. See [What You May Need to Know About Programmatically Expanding and Collapsing Nodes](#).

When a user selects or deselects a row or a node, a `SelectionEvent` is fired. You can use the `selectionListener` to programmatically respond to the event. See [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).

Example 12-1 The listView Component

```
<af:listView id="listView" binding="{editor.component}"
            var="item" varStatus="vs" partialTriggers="::pprLV"
            value="{demolistView.taskModel}"
            selection="multiple">
    <af:listItem id="lvi">
        <af:showPopupBehavior popupId="::ctxtMenu"
                            triggerType="contextMenu"/>
        <af:panelGroupLayout id="panelGroupLayout1"
                            layout="horizontal"
                            styleClass="AFStretchWidth">
            <af:panelGroupLayout id="panelGroupLayout2"
                                layout="horizontal"
                                inlineStyle="margin-left:20px; width:45px"
                                halign="center" valign="middle">
                <af:image rendered="{vs.index %6 ==1}"
                            source="/images/error.png" id="il"
                            shortDesc="Error at Line #{vs.index + 1}"/>
            </af:panelGroupLayout>
            <af:panelGroupLayout id="panelGroupLayout3" layout="horizontal"
                                inlineStyle="width:100%">
                <af:panelGroupLayout id="panelGroupLayout5"
                                    layout="vertical"
                                    inlineStyle="min-width:300px">
                    <af:outputText id="outputText1" value="{item.taskName}"
                                    styleClass="taskName"/>
                </af:panelGroupLayout>
            </af:panelGroupLayout>
        </af:listItem>
    </af:listView>
```

```

        <af:outputText id="outputText2"
            value="#{item.projectDesc}"
            styleClass="taskProjectDesc"/>
        <af:outputText id="outputText3" value="#{item.created}"
            styleClass="taskCreated"/>
    </af:panelGroupLayout>
</af:panelGroupLayout>
<af:panelGroupLayout id="panelGroupLayout4"
    layout="horizontal" halign="end"
    valign="middle"
    inlineStyle="margin-right:20px">
    <af:button id="cb1" text="Action"
        shortDesc="Click To Invoke Action for Item #{vs.index + 1}">
        <af:showPopupBehavior popupId="::popupDialog"
            alignId="cb1" align="afterStart"/>
    </af:button>
</af:panelGroupLayout>
</af:panelGroupLayout>
</af:listItem>
</af:.listView>

```

How to Display a Collection in a List

You use a `listView` component bound to a `CollectionModel` instance and one `listItem` component to create the list. If you want to display a simple parent-child hearers, you place a second `listItem` component in the `groupHeaderStamp` facet. You then add layout components and other text components to display the actual data.

To display a collection in a list:

1. Create a Java class for the model to which the list will be bound. If you want the list to display groups with headers, the model must extend the `org.apache.myfaces.trinidad.model.TreeModel` class. If not, it should extend the `org.apache.myfaces.trinidad.model.CollectionModel` class.

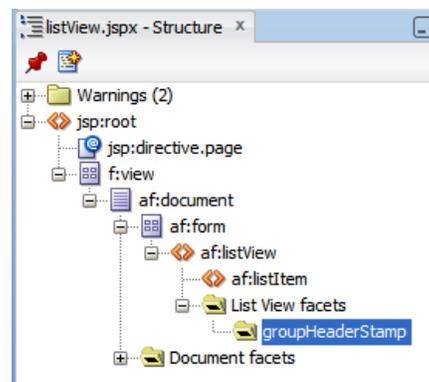
Tip:

You may also use other model classes, such as `java.util.List`, `array`, and `javax.faces.model.DataModel`. If you use one of these other classes, the `listView` component automatically converts the instance into a `CollectionModel` class, but without any additional functionality. For information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

2. In the Components window, from the Data Views panel, drag and drop a **List View** on to the page.
3. In the Properties window, expand the Other section and set the following:
 - **Value:** Specify an EL expression to bind the list to the mode created in Step 1.
 - **Var:** Specify a variable name to represent each node.
 - **First:** Specify a row to set as the first row to display in the list.
 - **FetchSize:** Set the size of the block that should be returned with each data fetch. The default is 25.

- **Rows:** Specific the number of rows to display in the range of rows. By default this is 25 (the same value as the `fetchSize` attribute).
 - **SelectedRowKeys:** Optionally enter the keys for the nodes that should be initially selected. See [What You May Need to Know About Programmatically Selecting Nodes](#).
 - **Selection:** Set a value to make the rows selectable (this is the `rowSelection` attribute). Valid values are: `none`, `single`, and `multiple`. For information about how to then programmatically perform some action on the selected rows, see [What You May Need to Know About Performing an Action on Selected Rows in Tables](#).
4. Drag and drop a **List Item** as a child to the `listView` component.
 5. Drag and drop layout and other components into the `listView` component, to create your desired configuration. See [Figure 12-33](#) and [Example 12-1](#) for an example.
 6. If you want the `listView` component to display a simple hierarchy, drag and drop a **List Item** into the `groupHeaderStamp` facet. [Figure 12-36](#) shows the `groupHeaderStamp` facet in the Structure window.

Figure 12-36 The `groupHeaderStamp` Facet in the Structure Window



7. Drag and drop an **Output Text** into the `listItem` component to display your header text, and configure the `outputText` component as needed.

What You May Need to Know About Scrollbars in a List View

Similar to tables, you can configure the `listView` component to use scrollbars. When you configure the `listView` component to use scrolling, in iOS operating systems, by default, the scrollbars only appear when you mouseover the content. Otherwise, they remain hidden. You can configure your application so that this same behavior occurs on other operating systems as well, by adding the `-tr-overflow-style: autohiding-scrollbar` skinning property to `af|listView` selector.

```
af|listView {
  -tr-overflow-style: autohiding-scrollbar
}
```

Note:

On operating systems other than iOS, initially set style attribute to `overflow:hidden` for listView root element. For the `mouseover` event, set style attribute to `overflow:auto` and for the `mouseout` event set style attribute back to `overflow:hidden`.

Displaying Images in a Carousel

You can display images in a revolving carousel structure using ADF `carouselItem` component. You can also configure the display of the carousel structure.

You can display images in a revolving carousel, as shown in [Figure 12-37](#). Users can change the image at the front either by using the slider at the bottom or by clicking one of the auxiliary images to bring that specific image to the front.

Figure 12-37 The ADF Faces Carousel



By default, the carousel displays horizontally. The objects within the horizontal orientation of the carousel are vertically-aligned to the middle and the carousel itself is horizontally-aligned to the center of its container.

You can configure the carousel so that it can be displayed vertically, as you might want for a reference Rolodex. By default, the objects within the vertical orientation of the carousel are horizontally-aligned to the center and the carousel itself is vertically aligned middle, as shown in [Figure 12-38](#). You can change the alignments using the carousel's alignment attributes.

Figure 12-38 Vertical Carousel Component



 **Best Practice:**

Generally the carousel should be placed in a parent component that stretches its children (such as a `panelSplitter` or `panelStretchLayout`). If you do not place the carousel in a component that stretches its children, your carousel will display at the default dimension of 500px wide and 300px tall. You can change these dimensions.

Instead of partially displaying the previous and next images, you can configure your carousel so that it displays images in a filmstrip design, as shown in [Figure 12-39](#), or in a roomy circular design, as shown in [Figure 12-40](#).

Figure 12-39 Carousel Filmstrip Display

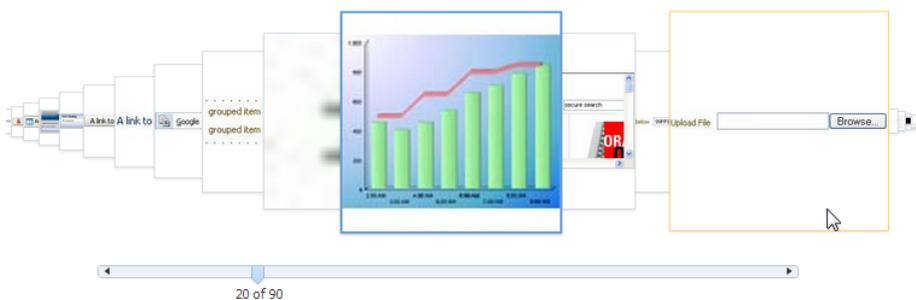


Figure 12-40 Carousel Roomy Circular Display



By default, when the carousel is configured to display in the circular mode, when you hover over an auxiliary item (that is, an item that is not the current item at the center), the item is outlined to show that it can be selected (note that this outline will only appear if your application is using the Skyros and above skins). You can configure the carousel so that instead, the item pops out and displays at full size, as shown in Figure 12-41.

Figure 12-41 Auxiliary Item Pops Out on Hover



You can also configure your carousel so that it displays only the current image, as shown in Figure 12-42.

Figure 12-42 Carousel Can Display Just One Image

You can configure the controls used to browse through the images. You can display a slider with next and previous arrows that spans more than one image, display only next and previous buttons, as shown in [Figure 12-42](#), or display next and previous buttons, along with the slide counter, as shown in [Figure 12-43](#).

Figure 12-43 Next and Previous Buttons Without a Slider

A child `carouselItem` component displays the objects in the carousel, along with a title for the object. Instead of creating a `carouselItem` component for each object to be displayed, and then binding these components to the individual object, you bind the `carousel` component to a complete collection. The component then repeatedly renders one `carouselItem` component by stamping the value for each item, similar to the way a tree stamps out each row of data. As each item is stamped, the data for the current item is copied into a property that can be addressed using an EL expression using the `carousel` component's `var` attribute. Once the carousel has completed rendering, this property is removed or reverted back to its previous value. Carousels contain a `nodeStamp` facet, which is both a holder for the `carouselItem` component used to display the text and short description for each item, and also the parent component to the image displayed for each item.

For example, the `carouselItem` JSF page in the ADF Faces Components Demo shown in [Figure 12-37](#) contains a `carousel` component that displays an image of each of the ADF Faces components. The `demoCarouselItem` (`CarouselBean.java`) managed bean contains a list of each of these components. The value attribute of the `carousel` component is bound to the `items` property on that bean, which represents that list. The `carousel` component's `var` attribute is used to hold the value for each item to display, and is used by both the `carouselItem` component and `image` component to

retrieve the correct values for each item. The following example shows the JSF page code for the carousel. For information about stamping behavior in a carousel, see [Displaying Data in Trees](#).

```
<af:carousel id="carousel" binding="#{editor.component}"
    var="item"
    value="#{demoCarousel.items}"
    carouselSpinListener="#{demoCarousel.handleCarouselSpin}">
  <f:facet name="nodeStamp">
    <af:carouselItem id="crslItem" text="#{item.title}" shortDesc="#{item.title}">
      <af:image id="img" source="#{item.url}" shortDesc="#{item.title}" />
    </af:carouselItem>
  </f:facet>
</af:carousel>
```

A `carouselItem` component stretches its sole child component. If you place a single image component inside of the `carouselItem`, the image stretches to fit within the square allocated for the item (as the user spins the carousel, these dimensions shrink or grow).

Best Practice:

The `image` component does not provide any geometry management controls for altering how it behaves when stretched. You should use images that have equal width and height dimensions in order for the image to retain its proper aspect ratio when it is being stretched.

The `carousel` component uses a `CollectionModel` class to access the data in the underlying collection. This class extends the JSF `DataModel` class and adds on support for row keys. In the `DataModel` class, rows are identified entirely by index. However, to avoid issues if the underlying data changes, the `CollectionModel` class is based on row keys instead of indexes.

You may also use other model classes, such as `java.util.List`, `array`, and `javax.faces.model.DataModel`. If you use one of these other classes, the `carousel` component automatically converts the instance into a `CollectionModel` class, but without any additional functionality. For information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

Note:

If your application uses the Fusion technology stack, you can create ADF Business Components over your data source that represent the items, and the model will be created for you. You can then declaratively create the carousel, and it will automatically be bound to that model. See the "Using the ADF Faces Carousel Component" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

The carousel components are virtualized, meaning not all the items that are there for the component on the server are delivered to and displayed on the client. You

configure the carousel to fetch a certain number of rows at a time from your data source. The data can be delivered to the component either immediately upon rendering, or lazily fetched after the shell of the component has been rendered. By default, the carousel lazily fetches data for the initial request. When a page contains one or more of these components, the page initially goes through the standard lifecycle. However, instead of the carousel fetching the data during that initial request, a special separate partial page rendering (PPR) request is run on the component, and the number of items set as the value of the fetch size for the carousel is then returned. Because the page has just been rendered, only the Render Response phase executes for the carousel, allowing the corresponding data to be fetched and displayed. When a user does something to cause a subsequent data fetch (for example spinning the carousel for another set of images), another PPR request is executed.

 **Performance Tip:**

You should use lazy delivery when the page contains a number of components other than a carousel. Using lazy delivery allows the initial page layout and other components to be rendered first before the data is available.

Use immediate delivery if the carousel is the only context on the page, or if the carousel is not expected to return a large set of items. In this case, response time will be faster than using lazy delivery (or in some cases, simply perceived as faster), as the second request will not go to the server, providing a faster user response time and better server CPU utilizations. Note however that only the number of items configured to be the fetch block will be initially returned. As with lazy delivery, when a user's actions cause a subsequent data fetch, the next set of items are delivered.

A slider control allows users to navigate through the collection. Normally the thumb on the slider displays the current object number out of the total number of objects, for example 6 of 20. When the total number of objects is too high to calculate, the thumb on the slider will show only the current object number. For example, say a carousel is used for a company's employee directory. By default the directory might show faces for every employee, but it may not know without an expensive database call that there are exactly 94,409 employees in the system that day.

You can use other components in conjunction with the carousel. For example, you can add a toolbar or menu bar, and to that, add buttons or menu items that allow users to perform actions on the current object.

How to Create a Carousel

To create a carousel, you must first create the data model that contains the images to display. You then bind a `carousel` component to that model and insert a `carouselItem` component into the `nodeStamp` facet of the carousel. Lastly, you insert an `image` component (or other components that contain an `image` component) as a child to the `carouselItem` component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Images in a Carousel](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

Create the data model that will provide the collection of images to display. The data model can be a `List`, `Array`, `DataModel` or `CollectionModel`. If the collection is anything other than a `CollectionModel`, the framework will automatically convert it to a `CollectionModel`. For information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

The data model should provide the following information for each of the images to be displayed in the carousel:

- URL to the images
- Title, which will be displayed below the image in the carousel
- Short description used for text displayed when the user mouses over the image

For examples, see the `CarouselBean.java` and the `CarouselMediaBean.java` classes in the ADF Faces Components Demo application.

To Create a Carousel:

1. In the Components window, from the Data Views panel, drag and drop a **Carousel** onto the page.

 **Best Practice:**

Place the carousel in a parent container that stretches its children.

2. In the Properties window, expand the Common section, and set the following:
 - **Orientation:** By default, the carousel displays horizontally. Select `vertical` if you want it to display vertically, as shown in [Figure 12-38](#). If you set it to `horizontal`, you must configure how the items line up using the `halign` attribute. If you set it to `vertical`, set how the items line up using the `valign` attribute.
 - **Halign:** Specify how you want items in a vertical carousel to display. Valid values are:
 - **center:** Aligns the items so that they have the same centerpoint. This is the default.
 - **end:** Aligns the items so that the right edges line up (when the browser is displaying a left-to-right language).
 - **start:** Aligns the items so that the left edges line up (when the browser is displaying a left-to-right language).
 - **Valign:** Specify how you want items in a horizontal carousel to display. Valid values are:
 - **middle:** Aligns the items so that they have the same middle point. This is the default.
 - **bottom:** Aligns the items so that the bottom edges line up.
 - **top:** Aligns the items so that the top edges line up.

- **Value:** Bind the carousel to the model.
3. Expand the Data section and set the following:
 - **Var:** Enter a variable that will be used in EL to access the individual item data.
 - **VarStatus:** Enter a variable that will be used in EL to access the status of the carousel. Common properties of `varStatus` include:
 - `model`: Returns the `CollectionModel` for the component.
 - `index`: Returns the zero-based item index.
 4. Expand the Appearance section and set the following:
 - **EmptyText:** Enter text that should display if no items are returned. If using a resource bundle, use the dropdown menu to choose **Select Text Resource**.
 5. If you do not place the carousel in a parent component that stretches its children, then the carousel will display at 500px wide by 300px tall. You can change these settings. To do so, expand the Style section, click the Layout tab, and set a height and width in pixels.
 6. Expand the Behavior section, and set the following:
 - **FetchSize:** Set the size of the block that should be returned with each data fetch.
 - **ContentDelivery:** Specify when the data should be delivered. When the `contentDelivery` attribute is set to `immediate`, items are fetched at the same time the carousel is rendered. If the `contentDelivery` attribute is set to `lazy`, items will be fetched and delivered to the client during a subsequent request.
 - **CarouselSpinListener:** Bind to a handler method that handles the spinning of the carousel when you need logic to be executed when the carousel spin is executed. The following example shows the handler method on the `CarouselBean` which redraws the detail panel when the spin happens.

```
public void handleCarouselSpin(CarouselSpinEvent event)
{
    RichCarousel carousel = (RichCarousel)event.getComponent();
    carousel.setRowKey(event.getNewItemKey());
    ImageInfo itemData = (ImageInfo)carousel.getRowData();
    _currentImageInfo = itemData;

    // Redraw the detail panel so that we can update the selected details.
    RequestContext rc = RequestContext.getCurrentInstance();
    rc.addPartialTarget(_detailPanel);
}
```
 7. Expand the Advanced section and set **CurrentItemKey**. Specify which item is showing when the carousel is initially rendered. The value should be (or evaluate to) the item's primary key in the `CollectionModel`.
 8. Expand the Other section and set the following:
 - **AuxiliaryOffset:** Enter a value to control the offset shift factor that a carousel item will have relative to its nearest item towards the current carousel item in circular `DisplayItems` mode.
 - **AuxiliaryPopOut:** Select **hover** so that mousing over an auxiliary item will render it at full size with the opaque overlay removed.
 - **AuxiliaryScale:** Enter a value to control the size scaling factor that a carousel item will have relative to its nearest item towards the current carousel item in

circular `DisplayItems` mode. A value of 1 means the auxiliary items will be the same size. A value less than 1 means the auxiliary items will become smaller the further they are from the current item. A value greater than 1 means the auxiliary items will become larger the further they are from the current item.

- **ControlArea:** Specify the controls used to browse through the carousel images. Valid values are:
 - **full:** A slider displays with built-in next and previous buttons, the current item text, and the image number. This is the default.
 - **compact:** Only the next and previous buttons and item text are displayed.
 - **small:** Next and previous buttons are displayed, along with the current item text, and the image number.
 - **none:** No slider is displayed.
- **DisplayItems:** Select **circular** to have the carousel display multiple images. Select **oneByOne** to have the carousel display one image at a time.

 **Tip:**

To achieve a roomy circular design for the `DisplayItems` in circular mode, set the `AuxiliaryScale` to 0.8 and `AuxiliaryOffset` 0.8. To display items in a filmstrip design in circular mode, set `AuxiliaryScale` to 1.0 and `AuxiliaryOffset` to 1.1.

9. In the Components window, from the Data Views panel, drag a **Carousel Item** to the `nodeStamp` facet of the `Carousel` component.

Bind the `CarouselItem` component's attributes to the properties in the data model using the variable value set on the carousel's `var` attribute. For example, if you use `item` as the value for the `var` attribute, the value of the `carouselItem`'s `text` attribute would be `item.title` (given that `title` is the property used to access the text used for the carousel items on the data model).

10. In the Components window, from the General Controls panel, drag an image and drop it as a child to the `carouselItem`.

Bind the `image` component's attributes to the properties in the data model using the variable value set on the carousel's `var` attribute. For example, if the carousel uses `item` as the value for the `var` attribute, the value of the `image`'s `source` attribute would be `item.url` (given that `url` is the property used to access the image).

You can surround the image component with other components if you want more functionality. For example, [Figure 12-44](#) shows a carousel whose images are surrounded by a `panelGroupLayout` component and that also uses a `clientListener` to call a JavaScript function to show a menu and a navigation bar.

Figure 12-44 Using a More Complex Layout in a Carousel



The following example shows the corresponding page code.

```
<af:carouselItem id="mainItem" text="#{item.title}" shortDesc="#{item.title}">
  <af:panelGroupLayout id="itemPgl" layout="vertical">
    <af:image id="mainImg" source="#{item.url}" shortDesc="#{item.title}"
      styleClass="MyImage">
      <af:clientListener method="handleItemOver" type="mouseOver"/>
      <af:clientListener method="handleItemDown" type="mouseDown"/>
      <af:showPopupBehavior triggerType="contextMenu" popupId="::itemCtx"/>
    </af:image>
    <af:panelGroupLayout id="overHead" styleClass="MyOverlayHeader"
      layout="vertical" clientComponent="true">
      <af:menuBar id="menuBar">
        <af:menu id="menu" text="Menu">
          <af:commandMenuItem id="menuItem1" text="Menu Item 1"/>
          <af:commandMenuItem id="menuItem2" text="Menu Item 2"/>
          <af:commandMenuItem id="menuItem3" text="Menu Item 3"/>
        </af:menu>
      </af:menuBar>
    </af:panelGroupLayout>
    <af:panelGroupLayout id="overFoot" styleClass="MyOverlayFooter"
      layout="vertical" clientComponent="true"
      valign="center">
      <af:panelGroupLayout id="footHorz" layout="horizontal">
        <f:facet name="separator">
          <af:spacer id="footSp" width="8"/>
        </f:facet>
        <af:link . . .
          />
        <af:outputText id="pageInfo" value="Page 1 of 1"/>
        <af:link . . .
          />
      </af:panelGroupLayout>
    </af:panelGroupLayout>
  </af:panelGroupLayout>
</af:carouselItem>
```

The following example shows the corresponding JavaScript.

```

function handleItemOver(uiInputEvent)
{
    var imageComponent = uiInputEvent.getCurrentTarget();
    var carousel = null;
    var componentParent = imageComponent.getParent();
    while (componentParent != null)
    {
        if (componentParent instanceof AdfRichCarousel)
        {
            carousel = componentParent;
        }
        componentParent = componentParent.getParent();
    }
    if (carousel == null)
    {
        AfLogger.LOGGER.severe("Unable to find the carousel component!");
        return;
    }
    var currentItemKeyPattern = ":"+ carousel.getCurrentItemKey() +":";

    var overlayHeaderComponent = imageComponent.findComponent("overHead");
    var overlayHeaderId = overlayHeaderComponent.getClientId();
    // In IE we get mouseover for other items as well. This is despite having an
    // overlay div on top
    if(overlayHeaderId.indexOf(currentItemKeyPattern) == -1)
    return;
    if (overlayHeaderId != window._myHeader)
    {
        // ensure only one set of overlays are visible
        hideExistingOverlays();
    }
    var overlayFooterComponent = imageComponent.findComponent("overFoot");

    window._myHeader = overlayHeaderComponent.getClientId();
    window._myFooter = overlayFooterComponent.getClientId();

    // do not propagate to the server otherwise all stamps will get this property on
    // next data fetch
    overlayHeaderComponent.setProperty("inlineStyle", "display:block", false,
    AdfUIComponent.PROPROPAGATE_LOCALLY);
    overlayFooterComponent.setProperty("inlineStyle", "display:block",
    false, AdfUIComponent.PROPROPAGATE_LOCALLY);
}

function handleItemDown(uiInputEvent)
{
    {
        if (uiInputEvent.isLeftButtonPressed())
        {
            // Only hide the overlays if the left button was pressed
            hideExistingOverlays();
        }
    }
}

```

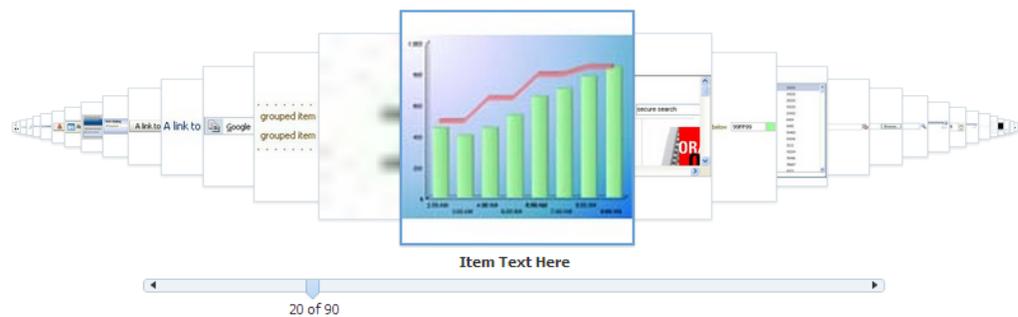
 **Performance Tip:**

The simpler the structure for the carousel, the faster it will perform.

What You May Need to Know About the Carousel Component and Different Browsers

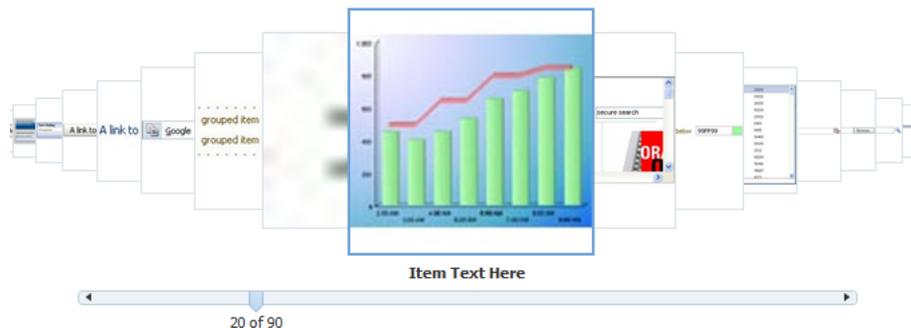
In some browsers, the visual decoration of the carousel's items will be richer. For example, Safari and Google Chrome display subtle shadows around the carousel's items, and the noncurrent items have a brightness overlay to help make clear that the auxiliary items are not the current item, as shown in [Figure 12-45](#).

Figure 12-45 Carousel Component Displayed in Google Chrome



[Figure 12-46](#) shows the same component in Internet Explorer.

Figure 12-46 Carousel Component Displayed in Microsoft Internet Explorer



Exporting Data from Table, Tree, or Tree Tables

You can export data from ADF Gantt chart to an Excel spreadsheet or you can export as a comma-separated values file. The formatting information is also exported along with data.

You can export the data from a table, tree, or tree table, or from a table region of the data visualization project Gantt chart to a Microsoft Excel spreadsheet or to a comma-separated values (CSV) file. To allow users to export a table, you create an action source, such as a button or link that will be used to invoke the export, and add an `exportCollectionActionListener` component and associate it with the data you wish to export. The formatting information is included in the exported Excel spreadsheet, so

that the data types are identified correctly and you can use the inbuilt functions of the MS Excel on various columns. However, if the cell format is not valid, then by default the text is exported as `Text` format. You can configure the `exportCollectionActionListener` so that all the rows of the source table or tree will be exported, or so that only the rows selected by the user will be exported.



Tip:

You can also export data from a DVT pivot table. See [Exporting from a Pivot Table](#).

For example, [Figure 12-47](#) shows the table from the ADF Faces Components Demo application that includes buttons that allow users to export the data to an Excel spreadsheet or as a CSV file.

Figure 12-47 Table with Buttons for Exporting Data

		Export All Rows to Excel		Export Selected Rows to Excel		Export All Rows to CSV		Export Selected Rows to CSV	
Number	Name	Size of the file in Kilo B	Number	Date Modified	Col5	Col6			
0	.	0 B	0	07/12/2004	.	07/12/2004			
1	..	0 B	1	07/12/2004	..	07/12/2004			
2	admin.jar	1 KB	2	05/11/2004	admin.jar	05/11/2004			
3	applib	0 B	3	07/12/2004	applib	07/12/2004			
4	applications	0 B	4	07/12/2004	applications	07/12/2004			
5	config	0 B	5	07/12/2004	config	07/12/2004			
6	connectors	0 B	6	07/12/2004	connectors	07/12/2004			
7	database	0 B	7	07/12/2004	database	07/12/2004			

When the user clicks a button, the listener processes the exporting of all the rows to a spreadsheet or CSV. As shown in [Figure 12-47](#), you can also configure the `exportCollectionActionListener` component so that only the rows the user selects are exported.

Only the following can be exported:

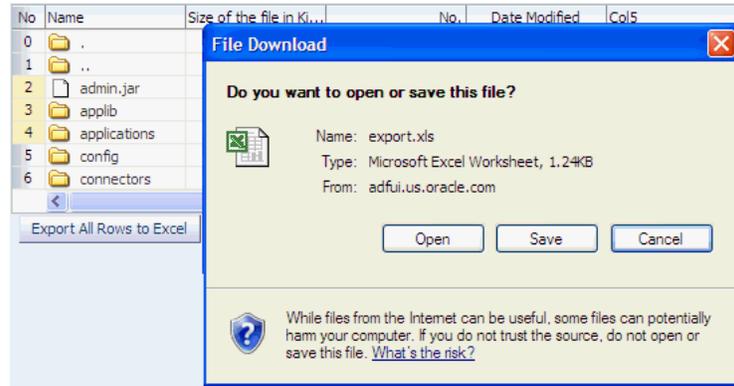
- Value of value holder components (such as `input` and `output` components).
- Value of `selectItem` components used in `selectOneChoice` and `selectOneListbox` components (the value of `selectItem` components in other selection components are not exported).
- Value of the `text` attribute of a command component.
- Value of the `shortDesc` attribute on `image` and `icon` components.

If you do not want the value of the `shortDesc` attribute on `image` and `icon` components to be exported (for example, if a cell contains an image), you can use the predefined `skipObjectComponent` filter method. This method will run before the `exportCollectionActionListener`, and will keep any `Object` components from being exported. You can also create your own custom filter method to apply any needed logic before the `exportCollectionActionListener` runs.

Depending on the browser, and the configuration of the listener, the browser will either open a dialog, allowing the user to either open or save the file as shown in [Figure 12-48](#), or the file will be displayed in the browser. For example, if the user is

viewing the page in Microsoft Internet Explorer, and no file name has been specified on the `exportCollectionActionListener` component, the file is displayed in the browser. In Mozilla Firefox, the dialog opens.

Figure 12-48 Exporting to Excel Dialog



If the user chooses to save the file, it can later be opened in a spreadsheet application, as shown in Figure 12-49. If the user chooses to open the file, what happens depends on the browser. For example, if the user is viewing the page in Microsoft Internet Explorer and is exporting a spreadsheet, the spreadsheet opens in the browser window. If the user is viewing the page in Mozilla Firefox, the spreadsheet opens in Excel.

Figure 12-49 Exported Data File in Excel

	A	B	C	D	E
1	No	Name	Size of the file in Kilo Bytes	No.	Date Modified
2	0	.	0 B	0	7/12/2004
3	1	..	0 B	1	7/12/2004
4	2	admin.jar	1 KB	2	5/11/2004
5	3	applib	0 B	3	7/12/2004
6	4	applications	0 B	4	7/12/2004
7	5	config	0 B	5	7/12/2004
8	6	connectors	0 B	6	7/12/2004
9	7	database	0 B	7	7/12/2004
10	8	default-web-app	0 B	8	7/12/2004
11	9	iiop.jar	1,290 KB	9	5/11/2004
12	10	iiop_gen_bin.jar	37 KB	10	5/11/2004
13	11	iiop_rmic.jar	144 KB	11	5/11/2004
14	12	jazn	0 B	12	7/12/2004
15	13	jazn.jar	266 KB	13	5/11/2004

 **Note:**

For spreadsheets, you may receive a warning from Excel stating that the file is in a different format than specified by the file extension. This warning can be safely ignored.

How to Export Table, Tree, or Tree Table Data to an External Format

You create a command component, such as a button, link, or menu item, and add the `exportCollectionActionListener` inside this component. Then you associate the data collection you want to export by setting the `exportCollectionActionListener` component's `exportedId` attribute to the ID of the collection component whose data you wish to export.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Exporting Data from Table, Tree, or Tree Table](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

You should already have a table, tree, or tree table on your page. If you do not, follow the instructions in this chapter to create a table, tree, or tree table. For example, to add a table, see [Displaying Data in Tables](#).

 **Tip:**

If you want users to be able to select rows to export, then configure your table to allow selection. See [Formatting Tables](#).

To export collection data to an external format:

1. In the Components window, from the General Controls panel, drag and drop a command component, such as a button, to your page.

 **Tip:**

If you want your table, tree, or tree table to have a toolbar that will hold command components, you can wrap the collection component in a `panelCollection` component. This component adds toolbar functionality. See [Displaying Table Menus, Toolbars, and Status Bars](#).

You may want to change the default label of the command component to a meaningful name such as **Export to a Spreadsheet**.

2. In the Components window, from the Operations panel, drag an **Export Collection Action Listener** as a child to the command component.
3. In the Insert Export Collection Action Listener dialog, set the following:
 - **ExportedId:** Specify the ID of the table, tree, or tree table to be exported. Either enter it manually or use the dropdown menu to choose **Edit**. Use the Edit Property dialog to select the component.
 - **Type:** Set to `excelHTML` to export to a Microsoft Excel spreadsheet. Set to `CSV` to export to a comma-separated values file.

4. With the `exportCollectionActionListener` component still selected, in the Properties window, set the following:
- **Filename:** Specify the proposed file name for the exported content. When this attribute is set, a "Save File" dialog will typically be displayed, though this is ultimately up to the browser. If the attribute is not set, the content will typically be displayed inline, in the browser, if possible.
 - **Title:** Specify the title of the exported document. Whether or not the title is displayed and how exactly it is displayed depends on the spreadsheet application.
 - **ExportedRows:** Specify if you want to export all rows in the table, or only rows selected by the user. If your table uses the `detailStamp` facet, you can elect to either export that data or not (for information about the `detailStamp` facet, see [Adding Hidden Capabilities to a Table](#)). Set to one of the following:
 - `all`: All rows will be exported. This includes the currently visible rows and all hidden rows that have not been disclosed in the component UI.
 - `allWithoutDetails`: All rows, except the data in the `detailStamp` facet, will be exported.
 - `disclosed`: Only the rows as currently displayed in the page will be exported. That includes disclosed (shown) nodes plus their leaf nodes.
 - `disclosedWithoutDetails`: Only the rows as currently displayed in the page will be exported, except for the data in the `detailStamp` facet.
 - `selected`: Only the rows the user has selected will be exported. Set only if the underlying collection supports selection.
 - `selectedWithoutDetails`: Only the rows the user has selected will be exported, except for the data in the `detailStamp` facet. Set only if the underlying collection supports selection.
 - **RowLimit:** Enter a number that represents the maximum number of rows that can be exported. Enter `-1` if there should be no limit.
 - **Charset:** By default, UTF-8 is used. You can specify a different character set if needed.
 - **FilterName:** Enter `skipObjectComponent`, if you want a built-in method to run that will skip any `Object` component from being processed.
 - **FilterMethod:** Enter an EL expression that evaluates to a method that will be invoked before the `ExportCollectionActionListener` that will handle any needed override logic.

The following example shows the code for a table and its `exportCollectionActionListener` component. Note that the `exportedId` value is set to the table `id` value.

```
<af:table contextMenuId="thePopup" selectionListener="#{fs.Table}"
  rowselection="multiple" columnselection="multiple"
  columnBandingInterval="1"
  binding="#{editor.component}" var="test1" value="#{tableTestData}"
  id="table" summary="table data">
  <af:column>
    . . .
  </af:column>
</af:table>
<af:button text="Export To Excel" immediate="true">
```

```
<af:exportCollectionActionListener type="excelHTML" exportedId="table"  
    filename="export.xls" title="ADF Faces Export"/>
```

What Happens at Runtime: How Row Selection Affects the Exported Data

Exported data is exported in index order, not selected key order. This means that if you allow selected rows to be exported, and the user selects rows (in this order) 8, 4, and 2, then the rows will be exported and displayed in Excel in the order 2, 4, 8.

Accessing Selected Values on the Client from Collection-Based Components

In the ADF framework, the client-support for EL is not available. To access row-specific data on the client, you must use EL on the collection-based component.

Since there is no client-side support for EL in the ADF Faces framework, nor is there support for sending entire collection models to the client, if you need to access values on the client using JavaScript, the client-side code cannot rely on component stamping to access the value. Instead of reusing the same component instance on each row, a new JavaScript component is created on each row (assuming any component needs to be created at all for any of the rows), using the fully resolved EL expressions.

Therefore, to access row-specific data on the client, you need to use the collection-based component itself to access the value. To do this without a client-side data model, you use a client-side selection change listener.

How to Access Values from a Selection in Stamped Components.

To access values on the client from a collection-based component, you first need to make sure the component has a client representation. Then you need to register a selection change listener on the client and then have that listener handle determining the selected row, finding the associated stamped component for that row, use the collection component to determine the row-specific name, and finally interact with the selected data as needed.

Before you begin:

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Collection-Based Components](#).

To access selected values from collection-based components:

1. In the Structure window for your page, select the component associated with the stamped row. For example, in the following example the table uses an `outputText` component to display the stamped rows.

```
<af:table var="row" value="#{data}" rowSelection="single">  
    <af:column headerText="Name">  
        <af:outputText value="#{row.name}"/>  
    </af:column>  
</af:table>
```

Set the following on the component:

- Expand the **Common** section of the Properties window and if one is not already defined, set a unique ID for the component using the `Id` attribute.
 - Expand the **Advanced** section and set **ClientComponent** to `True`.
2. In the Components window, from the Operations panel, drag and drop a **Client Listener** as a child to the collection component.
 3. In the Insert Client Listener dialog, enter a function name in the **Method** field (you will implement this function in the next step), and select `selection` from the **Type** dropdown.

If for example, you entered **mySelectedRow** as the function, JDeveloper would enter the code shown in bold in the following example.

```
<af:table var="row" value="#{data}" rowSelection="single">
  <af:clientListener type="selection" method="mySelectedRow"/>
  ...
</af:table>
```

This code causes the `mySelectedRow` function to be called any time the selection changes.

4. In your JavaScript library, implement the function entered in the last step. This function should do the following:
 - Figure out what row was selected. To do this, use the event object that is passed into the listener. In the case of selection events, the event object is of type `AdfSelectionEvent`. This type provides access to the newly selected row keys via the `getAddedSet()` method, which returns a POJSO (plain old JavaScript object) that contains properties for each selected row key. Once you have access to this object, you can iterate over the row keys using a "for in" loop. For example, the code in the following example extracts the first row key (which in this case, is the only row key).

```
function showSelectedName(event)
{
  var firstRowKey;
  var addRowKeys=event.getAddedSet();

  for(var rowKey in addedRowKeys)
  {
    firstRowKey=rowKey;
    break;
  }
}
```

- Find the stamped component associated with the selected row. The client-side component API `AdfUIComponent` exposes a `findComponent()` method that takes the ID of the component to find and returns the `AdfUIComponent` instance. When using stamped components, you need to find a component not just by its ID, but by the row key as well. In order to support this, the `AdfUITable` class provides an overloaded method of `findComponent()`, which takes both an ID as well as a row key.

In the case of selection events, the component is the source of the event. So you can get the collection component from the source of the event and then use the collection component to find the instance using the ID and row key. The following example shows this, where `nameStamp` is the ID of the table.

```
// The table needs to find the stamped component.
// Fortunately, in the case of selection events, the
```

```
// table is the event source.
var table = event.getSource();

// Use the table to find the name stamp component by id/row key:
var nameStamp = table.findComponent("nameStamp", firstRowKey);
```

5. Add any additional code needed to work with the component. Once you have the stamped component, you can interact with it as you would with any other component. For example, the following code shows how to use the stamped component to get the row-specific value of the `name` attribute and then display the name in an alert.

```
if (nameStamp)
{
    // This is the row-specific name
    var name = nameStamp.getValue();

    alert("The selected name is: " + name);
}
```

The following example shows the entire code for the JavaScript.

```
function showSelectedName(event)
{
    var firstRowKey;
    var addedRowKeys = event.getAddedSet();

    for (var rowKey in addedRowKeys)
    {
        firstRowKey = rowKey;
        break;
    }
    // The table needs to find the stamped component.
    // Fortunately, in the case of selection events, the
    // table is the event source.
    var table = event.getSource();

    // We use the table to find the name stamp component by id/row key:
    var nameStamp = table.findComponent("nameStamp", firstRowKey);

    if (nameStamp)
    {
        // This is the row-specific name
        var name = nameStamp.getValue();

        alert("The selected name is: " + name);
    }
}
```

What You May Need to Know About Accessing Selected Values

Row keys are tokenized on the server, which means that the row key on the client may have no resemblance to the row key on the server. As such, only row keys that are served up by the client-side APIs (like `AdfSelectionEvent.getAddedSet()`) are valid.

Also note that `AdfUITable.findComponent(id, rowKey)` method may return `null` if the corresponding row has been scrolled off screen and is no longer available on the client. Always check for `null` return values from `AdfUITable.findComponent()` method.

13

Using List-of-Values Components

This chapter describes how to use a list-of-values component to display a model-driven list of objects from which a user can select a value. It describes how to create a data model that uses the list-of-values functionality with the `ListOfValues` data model. It also describes how to add the `inputListOfValues` and `inputComboboxListOfValues` components to a page.

This chapter includes the following sections:

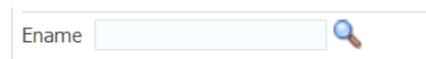
- [About List-of-Values Components](#)
- [Creating the ListOfValues Data Model](#)
- [Using the inputListOfValues Component](#)
- [Using the InputComboboxListOfValues Component](#)
- [Using the InputSearch Component](#)

About List-of-Values Components

The ADF List-of-Values (LOV) input component is a set of values populated either when the user starts typing the search value in the LOV field or when the user hits the submit button of the LOV field. This is helpful when you want to display the data to the users from one list item to another.

ADF Faces provides two list-of-values (LOV) input components that can display multiple attributes of each list item and can optionally allow the user to search for the needed item. These LOV components are useful when a field used to populate an attribute for one object might actually be contained in a list of other objects, as with a foreign key relationship in a database. For example, suppose you have a form that allows the user to edit employee information. Instead of having a separate page where the user first has to find the employee record to edit, that search and select functionality can be built into the form, as shown in [Figure 13-1](#).

Figure 13-1 List-of-Values Input Field



In this form, the employee name field is an LOV that contains a list of employees. When the user clicks the search icon of the `inputListOfValues` component, a Search and Select popup dialog displays all employees, along with a search field that allows the user to search for the employee, as shown in [Figure 13-2](#). If the results table is empty, you can display a custom message via the `resultTable` facet.

Figure 13-2 The Search Popup Dialog for a List-of-Values Component

The search popup dialog is titled "Search" and includes a "Basic" tab. It features an "Ename" search field containing the letter "A", and "Search" and "Reset" buttons. Below the search area is a table listing employee data from the EMP table. The table has columns for EMPNO, ENAME, JOB, MGR, and DEPTNO. The data is as follows:

EMPNO	ENAME	JOB	MGR	DEPTNO
0	Adam0	Engineer	1	10
1	Avance1	Manager	1	20
2	Abdul2	Analyst	1	10
3	Blake3	Technician	1	30
4	Bob4	Engineer	1	40
5	Brenta5	Manager	1	30
6	Bejond6	Analyst	1	10
7	Calvin7	Analyst	1	10
8	Carl8	Engineer	1	40
9	Chris9	Technician	1	20
10	Claire10	Manager	1	20
11	Dave11	Operator	1	40
12	David12	Manager	1	10
13	Derek13	Analyst	1	30
14	Eric14	Technician	1	10
15	Elane15	Engineer	1	40
16	Frank16	Analyst	1	50
17	Fonda17	Technician	1	30
18	Ford18	Manager	1	30
19	Gary19	Analyst	1	20
20	Good20	Engineer	1	60
21	Goodon21	Analyst	1	60
22	T.J22	Technician	1	50
23	James23	Engineer	1	10
24	Henry24	Operator	1	20

At the bottom of the dialog are "OK" and "Cancel" buttons.

When the user returns to the page, the current information for that employee is displayed in the form, as shown in Figure 13-3. The user can then edit and save the data.

Figure 13-3 Form Populated Using LOV Component

The form displays the following data:

Ename	Avance118
Empno	118
Deptno	20
HireDate	1/15/2002
Manager	1
Salary	24232
Commision	32211

As shown in the preceding figures, the `inputListOfValues` component provides a popup dialog from which the user can search for and select an item. The list is displayed in a table. In contrast, the `inputComboboxListOfValues` component allows the user two different ways to select an item to input: from a simple dropdown list, or by searching as you can in the `inputListOfValues` component.

You can also create custom content to be rendered in the Search and Select dialog by using the `searchContent` facet. You define the `returnPopupDataValue` attribute and programmatically set it with a value when the user selects an item from the Search and Select dialog and then closes the dialog. This value will be the return value from the `ReturnPopupEvent` to the `returnPopupListener`. When you implement the `returnPopupListener`, you can perform functions such as setting the value of the LOV component and its dependent components, and displaying the custom content. In the `searchContent` facet you can add components such as tables, trees, and input text to display your custom content.

If you implement both the `searchContent` facet and the `ListOfValues` model, the `searchContent` facet implementation will take precedence in rendering the Search and Select dialog. The following example shows the code to display custom content using a table component.

```
<af:inputListOfValues model="#{bean.listOfValuesModel}"
...
    returnPopupDataValue="#{bean.returnPopupDataValue}"
    returnPopupListener="#{bean.returnPopupListener}">
  <f:facet name="searchContent">
    <af:table id="t1" value="#{bean.listModel}" var="row"
      selectionListener="#{bean.selected}"
    ...
  </f:facet>
</af:inputListOfValues>
```

Both components support the auto-complete feature, which allows the user to enter a partial value in the input field, tab out (or click out), and have the dialog populated with one or more rows that match the partial criteria. For auto-complete to work, you must implement logic so that when the user tabs or clicks out after a partial entry, the entered value is posted back to the server. On the server, your model implementation filters the list using the partially entered value and performs a query to retrieve the list of values. ADF Faces provides APIs for this functionality. To configure the LOV input component to auto-complete using only the Search and Select dialog when no unique matching row is available, you need to add the following element to the `adf-config.xml` file:

You can implement the LOV component to carry forward the value entered in the input field of `inputListOfValues` and `inputComboboxListOfValues` to the appropriate search field in the search dialog. You can use the `setCriterionValue(Object Value)` API method, which sets the search field's value and also changes the operator of the search field.

```
public void setCriterionValue(Object value)
```

This method is invoked during the launch of advanced search panel and sets the value of the criterion that matches the LOVs display attribute.

If you want to add the auto-complete feature when the user tabs or clicks out after entering a partial entry, you will need to disable the custom popup. In your `LaunchPopupListener()` code, add `launchPopupEvent.setLaunchPopup(false)` to

prevent the custom popup from launching when the user tabs or clicks out. Clicking on the Search link will still launch the Search and Select dialog. The following example shows the listener code in a managed bean that is used to disable the custom popup.

```
public void LaunchPopupListener(LaunchPopupEvent launchPopupEvent) {
    if (launchPopupEvent.getPopupType().equals
        (LaunchPopupEvent.PopupType.SEARCH_DIALOG)
        {
            ...

            launchPopupEvent.setLaunchPopup(false);
        }
}
```

In the situation where the user tabs out and no unique row match is available to auto-complete a partial input criteria (because duplicate values exist), the Search and Select dialog is displayed. When the user clicks out with a partial value and no unique row match is available, the user gets a validation error notifying them of duplicate values. In this case, you can optionally configure the Search and Select dialog to launch so the behavior for tab out and click out on duplicate values is the same (by default the dialog does not display for the click out action). To configure auto-complete to always use the Search and Select dialog when no unique matching row is available, you can add the following element to the `adf-config.xml` file

```
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/config">
    <lov-show-searchdialog-onerror>true</lov-show-searchdialog-onerror>
</adf-faces-config>
```

If the `readOnly` attribute is set to `true`, the input field is disabled. If `readOnly` is set to `false`, then the `editMode` attribute determines which type of input is allowed. If `editMode` is set to `select`, the value can be entered only by selecting from the list. If `editMode` is set to `input`, then the value can also be entered by typing.

You can also implement the LOV component to automatically display a list of suggested items when the user types in a partial value. For example, when the user enters `Ad`, then a suggested list which partially matches `Ad` is displayed as a suggested items list, as shown in [Figure 13-4](#). If there are no matches, a "No results found." message will be displayed.

Figure 13-4 Suggested Items List for an LOV



The user can select an item from this list to enter it into the input field, as shown in [Figure 13-5](#).

Figure 13-5 Suggested Items Selected

You add the *auto-suggest behavior* by adding the `af:autoSuggestBehavior` tag inside the LOV component with the tag's `suggestItems` values set to a method that retrieves and displays the list. You can create this method in a managed bean. If you are using ADF Model, the method is implemented by default.

In your LOV model implementation, you can implement a **smart list** that filters the list further. You can implement a smart list for both LOV components. If you are using ADF Model, the `inputComboboxListOfValues` allows you declaratively select a smart list filter defined as a view criteria for that LOV. If the smart list is implemented, and auto-suggest behavior is also used, auto-suggest will search from the smart list first. If the user waits for two seconds without a gesture, auto-suggest will also search from the full list and append the results. The `maxSuggestedItems` attribute specifies the number of items to return (-1 indicates a complete list). If `maxSuggestedItems > 0`, a **More** link is rendered for the user to click to launch the LOV's Search and Select dialog. The following example shows the code for an LOV component with both auto-suggest behavior and a smart list.

```
af:autoSuggestBehavior
  suggestItems="#{bean.suggestItems}"
  smartList="#{bean.smartList}"/>
  maxSuggestedItems="7"/>
```

Figure 13-6 shows how a list can be displayed by an `inputComboboxListOfValues` component. If the popup dialog includes a query panel or smart list is not being used a **Search** link is displayed at the bottom of the dropdown list. If a query panel is not used or if smart list is enable, a **More** link is displayed.

Figure 13-6 InputComboboxListOfValues Displays a List of Employee Names

The screenshot shows a web component with a search box labeled 'Ename' and a dropdown arrow. Below the search box is a list of 25 rows, each representing an employee. The first row is highlighted in blue. The list includes columns for an index, employee name, job title, and a value of 1. A search bar with the text 'Search...' is located at the bottom of the list.

Index	Name	Job Title	Value
0	Adam0	Engineer	1
1	Avance1	Manager	1
2	Abdul2	Analyst	1
3	Blake3	Technician	1
4	Bob4	Engineer	1
5	Brenta5	Manager	1
6	Bejond6	Analyst	1
7	Calvin7	Analyst	1
8	Carl8	Engineer	1
9	Chris9	Technician	1
10	Claire10	Manager	1
11	Dave11	Operator	1
12	David12	Manager	1
13	Derek13	Analyst	1
14	Eric14	Technician	1
15	Eilane15	Engineer	1
16	Frank16	Analyst	1
17	Fonda17	Technician	1
18	Ford18	Manager	1
19	Gary19	Analyst	1
20	Good20	Engineer	1
21	Goodon21	Analyst	1
22	T.J22	Technician	1
23	James23	Engineer	1
24	Henry24	Operator	1

Search...

You can control when the contents of the dropdown list are sent and rendered to the client using the `contentDelivery` attribute. When set to `immediate` delivery, the contents of the list are fetched during the initial request. With `lazy` delivery, the page initially goes through the standard lifecycle. However, instead of fetching the list content during that initial request, a special separate partial page rendering (PPR) request is run, and the list content is then returned. You can configure the list so that its contents are not rendered to the client until the first request to disclose the content and the contents then remain in the cache (`lazy`), or so that the contents are rendered each time there is a request to disclose them (`lazyUncached`), which is the default.

How you set the `contentDelivery` attribute effects when the `LaunchPopupEvent` is queued. When `contentDelivery` is set to `lazyUncached`, this event is queued while displaying the dropdown panel. When `contentDelivery` is `lazy`, the event is queued only the first time the dropdown displays. When set to `immediate`, the `LaunchPopupEvent` is not queued at all.

The dropdown list of the `inputComboboxListOfValues` component can display the following:

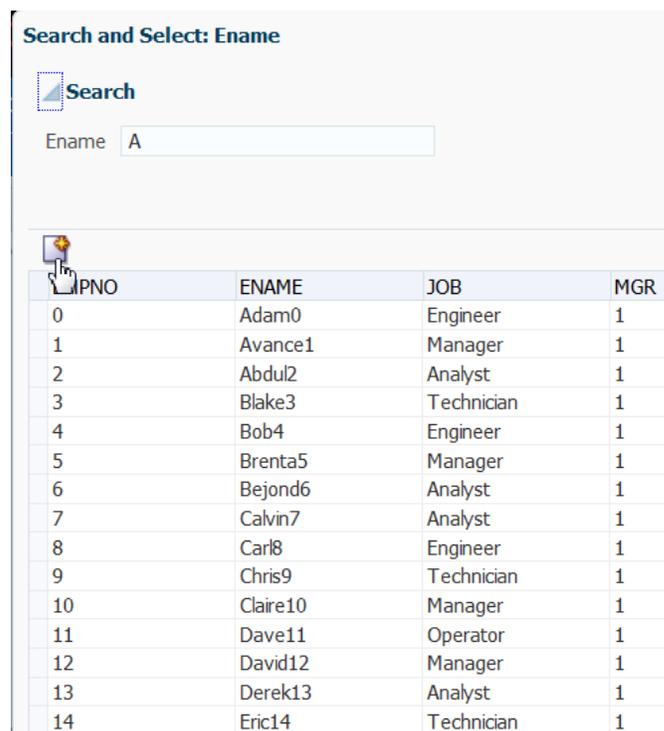
- **Full list:** As shown in [Figure 13-6](#), a complete list of items returned by the `ListOfValuesModel.getItems()` method.
- **Favorites list:** A list of recently selected items returned by the `ListOfValuesModel.getRecentItems()` method.

- Search link: A link that opens a popup Search and Select dialog. The link is not on the scrollable region on the dropdown list.
- `customActions` facet: A facet for adding additional content. Typically, this contains one or more `link` components. You are responsible for implementing any logic for the `link` to perform its intended action, for example, launching a popup dialog.

The number of columns to be displayed for each row can be retrieved from the model using the `getItemDescriptors()` method. The default is to show all the columns.

The popup dialog from within an `inputListOfValues` component or the optional search popup dialog in the `inputComboboxListOfValues` component also provides the ability to create a new record. For the `inputListOfValues` component, when the `createPopupId` attribute is set on the component, a `toolbar` component with a button is displayed with a create icon. At runtime, a `button` component appears in the LOV popup dialog, as shown in [Figure 13-7](#).

Figure 13-7 Create Icon in Toolbar of Popup Dialog



When the user clicks the **Create** button, a popup dialog is displayed that can be used to create a new record. For the `inputComboboxListOfValues`, instead of a toolbar, a link with the label **Create** is displayed in the `customActions` facet, at the bottom of the dialog. This link launches a popup where the user can create a new record. In both cases, you must provide the code to actually create the new record.

Both the `inputListOfValues` and the `inputComboboxListOfValues` components support the `context` facet. This facet allows you to add the `af:contextInfo` control, which can be used to show contextual information. When the user clicks in this area, it launches a popup window displaying contextual information.

 **Tip:**

Instead of having to build your own create functionality, you can use ADF Business Components and ADF data binding. See *Creating an Input Table in Developing Fusion Web Applications with Oracle Application Development Framework*.

Like the query components, the LOV components rely on a data model to provide the functionality. This data model is the `ListOfValuesModel` class. This model uses a table model to display the list of values, and can also access a query model to perform a search against the list. You must implement the provided interfaces for the `ListOfValuesModel` in order to use the LOV components.

 **Tip:**

Instead of having to build your own `ListOfValuesModel` class, you can use ADF Business Components to provide the needed functionality. See *Creating Databound Selection Lists and Shuttles in Developing Fusion Web Applications with Oracle Application Development Framework*.

When the user selects an item in the list, the data is returned as a list of objects for the selected row, where each object is the `rowData` for a selected row. The list of objects is available on the `ReturnPopupEvent` event, which is queued after a selection is made.

If you choose to also implement a `QueryModel` class, then the popup dialog will include a `Query` component that the user can use to perform a search and to filter the list. Note the following about using the `Query` component in an LOV popup dialog:

- The saved search functionality is not supported.
- The `Query` component in the popup dialog and its functionality is based on the corresponding `QueryDescriptor` class.
- The only components that can be included in the LOV popup dialog are `query`, `toolbar`, and `table`.

When the user clicks the **Search** button to start a search, the `ListOfValuesModel.performQuery()` method is invoked and the search is performed. For more information about the query model, see [Using Query Components](#).

You should use the list-of-values components when you have a more complex selection process that cannot be handled by the simpler select components. With list-of-values components, you can filter the selection list using accessors, smart list, auto-suggest, and other features to fine-tune the list criteria. You can create custom content in the popup window. You can add code to the `returnPopupListener` to perform functions when the popup window closes. A `customActions` facet can be used to add additional content. A create feature allows the user to create a new record. The list-of-values components offer a rich set of data input features for easier data entry.

Additional Functionality for List-of-Values Components

You may find it helpful to understand other ADF Faces features before you implement your list-of-values components. Additionally, once you have added a list-of-value component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that input components can use.

- **Client components:** Components can be client components. To work with the components on the client, see [Using ADF Faces Client-Side Architecture](#) .
- **JavaScript APIs:** All list-of-value components have JavaScript client APIs that you can use to set or get property values. See the *JavaScript API Reference for Oracle ADF Faces*.
- **Events:** List-of-value components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- You can add validation and conversion to list-of-values components. See [Validating and Converting Input](#).
- You can display tips and messages, as well as associate online help with list-of-values components. See [Displaying Tips, Messages, and Help](#).
- There may be times when you want the certain list-of-values components to be validated before other components on the page. See [Using the Immediate Attribute](#).
- You may want other components on the page to update based on selections you make from a list-of-values component. See [Using the Optimized Lifecycle](#).
- You can change the appearance of the components using skins. See [Customizing the Appearance Using Styles and Skins](#).
- You can make your list-of-values components accessible. See [Developing Accessible ADF Faces Pages](#).
- Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- The LOV components use the query component to populate the search list. For information on the query component, see [Using Query Components](#).
- Other list components, such as `selectOneChoice`, also allow users to select from a list, but they do not include a popup dialog and they are intended for smaller lists. For information about select choice components, list box components, and radio buttons, see [Using Input Components and Defining Forms](#) .
- If your application uses ADF Model, then you can create automatically bound forms using data controls (whether based on ADF Business Components or other business services). See *Creating a Basic Databound Page in Developing Fusion Web Applications with Oracle Application Development Framework*.

Creating the ListOfValues Data Model

A ListOfValues data model is a collection of interface classes. The ListOfValues data model provides you methods to retrieve a QueryModel and a TableModel to display a query component or a table.

Before you can use the LOV components, you must have a data model that uses the ADF Faces API to access the LOV functionality. For information on the LOV data model, see the *Java API Reference for Oracle ADF Faces*.

How to Create the ListOfValues Data Model

Before you begin:

It may be helpful to have an understanding of the list-of-values data model. See [Creating the ListOfValues Data Model](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for List-of-Values Components](#).

To create a ListOfValues model and associated events:

1. Create implementations of each of the interface classes. [Table 13-1](#) provides a description of the APIs.

Table 13-1 ListOfValues Model API

Method	Functionality
<code>autoCompleteValue()</code>	Called when the search icon is clicked or the value is changed and the user presses the Enter key or either tabs out or clicks out from the input field, as long as <code>autoSubmit</code> is set to <code>true</code> on the component. This method decides whether to open the dialog or to auto-complete the value. The method returns a list of filtered objects.
<code>valueSelected(value)</code>	Called when the value is selected from the Search and Select dialog and the OK button is clicked. This method gives the model a chance to update the model based on the selected value.
<code>isAutoCompleteEnabled()</code>	Returns a boolean to decide whether or not the auto complete is enabled.
<code>getTableModel()</code>	Returns the implementation of the <code>TableModel</code> class, on which the table in the search and select dialog will be based and created.
<code>getItems()</code> and <code>getRecentItems()</code>	Return the <code>items</code> and <code>recentItems</code> lists to be displayed in the combobox dropdown. Valid only for the <code>inputComboboxListOfValues</code> component. Returns null for the <code>inputListOfValues</code> component.
<code>getItemDescriptors()</code>	Return the list of <code>columnDescriptors</code> to be displayed in the dropdown list for an <code>inputComboboxListOfValues</code> component.

Table 13-1 (Cont.) ListOfValues Model API

Method	Functionality
<code>getQueryModel()</code> and <code>getQueryDescriptor()</code>	Return the <code>queryModel</code> based on which the query component inside the Search and Select dialog is created.
<code>performQuery()</code>	Called when the search button in the query component is clicked.
<code>setCriterionValue(Object value)</code>	Sets the value of the criterion that matches the LOV's display attribute in the advanced search panel. Default implementation does nothing. Object Value is the <code>submittedValue</code> of the component..

For an example of a `ListOfValues` model, see the `DemoLOVBean` and `DemoComboboxLOVBean` classes located in the `oracle.adfdemo.view.lov` package, found in the Application Sources directory of the ADF Faces application.

- For the `inputListOfValues` component, provide logic in a managed bean (it can be the same managed bean used to create your LOV model) that accesses the attribute used to populate the list. The `inputComboboxListOfValues` component uses the `getItems()` and `getRecentItems()` methods to return the list.
- For the Search and Select popup dialog used in the `InputListOfValues` component, or if you want the `InputComboboxListOfValues` component to use the Search and Select popup dialog, implement the `ListOfValuesModel.autoCompleteValue()` and `ListOfValuesModel.valueSelected()` methods. These methods open the popup dialog and apply the selected values onto the component.

Using the `inputListOfValues` Component

Using the `inputListOfValues` component you can select from list of values to help populate the LOV field on a page. You can use this component when the list of values to display is too large.

The `inputListOfValues` component uses the `ListOfValues` model you implemented to access the list of items, as documented in [Creating the ListOfValues Data Model](#).

How to Use the `InputListOfValues` Component

Before you begin:

It may be helpful to have an understanding of the `inputListOfValues` component. See [Using the `inputListOfValues` Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for List-of-Values Components](#).

You will need to complete this task:

Create a page or page fragment. If you also implemented the search API in the model, the component would also allows the user to search through the list for the value.

To add an `inputListOfValues` component:

1. In the Components window, from the Text and Selection panel, drag an **Input List Of Values** and drop it onto the page.
2. In the Properties window, expand the **Common** section and set the following attributes:
 - **model**: Enter an EL expression that resolves to your `ListOfValuesModel` implementation, as created in [How to Create the ListOfValues Data Model](#).
 - **value**: Enter an EL expression that resolves to the attribute values used to populate the list, as created in [How to Create the ListOfValues Data Model](#).
3. Expand the **Appearance** section and set the following attribute values:
 - **popupTitle**: Specify the title of the Search and Select popup dialog.
 - **searchDesc**: Enter text to display as a mouseover tip for the component.
 - **Placeholder**: Specify the text that appears in the `inputListOfValues` component if the component is empty and does not have focus. When the component gets focus, or has a value, then the placeholder text is hidden.

The placeholder text is used to inform the user what should be entered in the `inputListOfValues` component.

 **Note:**

The placeholder value will only work on browsers that fully support HTML5.

The rest of the attributes in this section can be populated in the same manner as with any other input component. See [Using the inputText Component](#).

4. Expand the **Behavior** section and set the following attribute values:
 - **autoSubmit**: Set to `true` if you want the component to automatically submit the enclosing form when an appropriate action takes place (a click, text change, and so on). This will allow the auto-complete feature to work.
 - **createPopupId**: If you have implemented a popup dialog used to create a new object in the list, specify the ID of that popup component. Doing so will display a `toolbar` component above the table that contains a `button` component bound to the popup dialog you defined. If you have added a dialog to the popup, then it will intelligently decide when to refresh the table. If you have not added a dialog to the popup, then the table will be always refreshed.
 - **launchPopupListener**: Enter an EL expression that resolves to a `launchPopupListener` that you implement to provide additional functionality when the popup is launched.
 - **returnPopupListener**: Enter an EL expression that resolves to a `returnPopupListener` component that you implement to provide additional functionality when the value is returned.
 - **Usage**: Specify how the `inputListOfValues` component will be rendered in HTML 5 browser. The valid values are `auto`, `text`, and `search`. Default is `auto`.

If the usage type is `search`, the `inputListOfValues` component will render as an HTML 5 search input type. Some HTML 5 browsers may add a **Cancel** icon that can be used to clear the search text.

The rest of the attributes in this section can be populated in the same manner as with any other input component. See [Using the `inputText` Component](#).

5. If you want users to be able to create a new item, create a popup dialog with the ID given in Step 4. See [Using Popup Dialogs, Menus, and Windows](#).
6. In the Components window, from the Operations panel, in the Behavior group, drag an **Auto Suggest Behavior** and drop it as a child to the `inputListOfValues` component.

If you add auto suggest behavior, you must not set the `immediate` property to `true`. Setting `immediate` to `true` will cause validation to occur in the Apply Request Values phase and any validation errors may suppress the displaying of the suggestion list.

7. In the Properties window, for each of the auto-suggest attributes, enter the:

- EL expression that resolves to the `suggestItems` method.

The method should return `List<javax.model.SelectItem>` of the `suggestItems`. The method signature should be of the form

```
List<javax.model.SelectItem>  
suggestItems( javax.faces.context.FacesContext,  
oracle.adf.view.rich.model.AutoSuggestUIHints)
```

- EL expression that resolves to the `smartList` method. The method should return `List<javax.model.SelectItem>` of the smart list items.
- You can modify the behavior of `af:autoSuggestBehavior` to start to show the suggest items only when a user enters more than N characters. For example, if `minimumChar="3"`, then the suggest list gets displayed only when you enter 3 or more characters.
- Number of items to be displayed in the auto-suggest list. Enter -1 to display the complete list.

If you are implementing this method in a managed bean, the JSF page entry should have the format shown in the following example

```
<af:inputListOfValues value="#{bean.value}" id="inputId">  
  ...  
  <af:autoSuggestBehavior  
    suggestItems="#{bean.suggestItems}"  
    smartList="#{bean.smartList}"  
    minimumChar="3"  
    maxSuggestedItems="7"/>  
</af:inputListOfValues>
```

```
<af:inputListOfValues value="#{bean.value}" id="inputId">
```

If the component is being used with a data model such as ADF Model, the `suggestItem` method should be provided by the default implementation.

8. If you are not using ADF Model, create the `suggestItems` method to process and display the list. The `suggestItems` method signature is shown in the following example.

```
List<javax.model.SelectItem> suggestItems( javax.faces.context.FacesContext,  
oracle.adf.view.rich.model.AutoSuggestUIHints)
```

What You May Need to Know About Dynamically Creating Auto Suggest Behavior for LOV Components

When you create an LOV dynamically using `af:query`, you cannot add **AutoSuggestBehavior** to the dynamically created LOV component. To achieve this, you need to first create the auto suggest behavior dynamically, configure it, and then add the dynamically created auto suggest behavior to the LOV component that is dynamically created. You can use the `af:query` component to create the behavior and to attach it to the dynamically created LOV component.

To create an auto suggest behavior, use the `AutoSuggestBehaviorTag` `behaviorTag = new AutoSuggestBehaviorTag()` in the `af:query` component.

To configure the properties of the dynamically created auto suggest behavior, use the `AutoSuggestBehaviorConfig` abstract class, which provides an instance that has all the configurations of **AutoSuggestBehavior**. The `AttributeDescriptor` API of `af:query` has a `getAutoSuggestBehaviorConfig` method that has the **AutoSuggestBehavior** properties. The query component invokes the `AttributeDescriptor` API to read the properties and set it on the `AutoSuggestBehaviorTag` instance that you created. To get the properties from **AutoSuggestBehavior**, use the following code:

```
AutoSuggestBehaviorConfig config =
attributeDescriptorInstance.getAutoSuggestBehaviorConfig();
behaviorTag.setMinChars(config.getMinChars());
behaviorTag.setSuggestItems(config.getSuggestItems());
```

To add the dynamically created behavior to the LOV component, use the following code:

```
FacesBean bean = component.getFacesBean();
PropertyKey key = bean.getType().findKey("clientListeners");
ClientListenerSet cls = (ClientListenerSet) bean.getProperty(key);
if (cls == null)
    cls = new ClientListenerSet();
behaviorTag.updateClientListenerSet(component, cls);
```

What You May Need to Know About Skinning the Search and Select Dialogs in the LOV Components

By default, the search and select dialogs that the `InputComboboxListOfValues` and `InputListOfValues` components can be resized by end users when they render. You can disable the end user's ability to resize these dialogs by setting the value of the `-tr-stretch-search-dialog` selector key to `false` in your application's skin file, as shown in the following example. The default value of the `-tr-stretch-search-dialog` selector key is `true`. For more information about skinning, see the skinning chapter.

```
af|inputComboboxListOfValues{
    -tr-stretch-search-dialog: false;
}
af|inputListOfValues{
    -tr-stretch-search-dialog: false;
}
```

Using the InputComboboxListOfValues Component

Using the `inputComboboxListOfValues` component you can select from a list of values to populate the LOV field on a page. The component also allows you to add a search link at the bottom of the populated list dropdown panel to launch the Search and Select dialog.

The `inputComboboxListOfValues` component allows a user to select a value from a dropdown list and populate the LOV field, and possibly other fields, on a page, similar to the `inputListOfValues` component. However, it also allows users to view the values in the list either as a complete list, or by most recently viewed. You can also configure the component to perform a search in a popup dialog, as long as you have implemented the query APIs, as documented in [Creating the ListOfValues Data Model](#).

For information about skinning and the Search and Select dialog sizing, see [What You May Need to Know About Skinning the Search and Select Dialogs in the LOV Components](#).

How to Use the InputComboboxListOfValues Component

Before you begin:

It may be helpful to have an understanding of the `inputComboboxListOfValues` component. See [Using the InputComboboxListOfValues Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for List-of-Values Components](#).

To add an `inputComboboxListOfValues` component:

1. In the Components window, from the Text and Selection panel, drag an **Input Combobox List Of Values** and drop it onto the page.
2. In the Properties window, expand the **Common** section and set the following attributes:
 - **model**: Enter an EL expression that resolves to your `ListOfValuesModel` implementation, as created in [How to Create the ListOfValues Data Model](#).
 - **value**: Enter an EL expression that resolves to the attribute values used to populate the list, as created in [How to Create the ListOfValues Data Model](#).
3. Expand the **Appearance** section and set the following attribute values:
 - **popupTitle**: Specify the title of the Search and Select popup dialog.
 - **searchDesc**: Enter text to display as a mouseover tip for the component.
 - **Placeholder**: Specify the text that appears in the `inputComboboxListOfValues` component if the component is empty and does not have focus. When the component gets focus, or has a value, then the placeholder text is hidden.

The placeholder text is used to inform the user what should be entered in the `inputComboboxListOfValues` component.

 **Note:**

The placeholder value will only work on browsers that fully support HTML5.

The rest of the attributes in this section can be populated in the same manner as with any other input component. See [Using the inputText Component](#).

4. Expand the **Behavior** section and set the following attribute values:
 - **autoSubmit**: Set to `true` if you want the component to automatically submit the enclosing form when an appropriate action takes place (a click, text change, and so on). This will allow the auto complete feature to work.
 - **createPopupId**: If you have implemented a popup dialog used to create a new object in the list, specify the ID of that popup component. Doing so will display a `toolbar` component above the table that contains a `button` component bound to the dialog you defined. If you have added a dialog to the popup, then it will intelligently decide when to refresh the table. If you have not added a dialog to the popup, then the table will always be refreshed.
 - **launchPopupListener**: Enter an EL expression that resolves to a `launchPopupListener` handler that you implement to provide additional functionality when the popup dialog is opened.
 - **returnPopupListener**: Enter an EL expression that resolves to a `returnPopupListener` handler that you implement to provide additional functionality when the value is returned.
 - **Usage**: Specify how the `inputComboboxListOfValues` component will be rendered in HTML 5 browser. The valid values are `auto`, `text`, and `search`. Default is `auto`.

If the usage type is `search`, the `inputComboboxListOfValues` component will render as an HTML 5 search input type. Some HTML 5 browsers may add a **Cancel** icon that can be used to clear the search text.

The rest of the attributes in this section can be populated in the same manner as with any other input component. See [Using the inputText Component](#).

5. If you want to control when the contents of the dropdown are delivered, expand the **Advanced** section, and set the `contentDelivery` attribute to one of the following:
 - `immediate`: All undisclosed content is sent when the dropdown list is disclosed. The `LaunchPopupEvent` will not be queued.
 - `lazy`: The undisclosed content is sent only when the content is first disclosed. Once disclosed and the content is rendered, it remains in memory. The `LaunchPopupEvent` will be queued only the first time the dropdown displays.
 - `lazyUncached`: The undisclosed content is created every time it is disclosed. If the content is subsequently hidden, it is destroyed. The `LaunchPopupEvent` will be queued while displaying the dropdown panel. This is the default.
6. If you are using a `launchPopupListener`, you can use the `getPopupType()` method of the `LaunchPopupEvent` class to differentiate the source of the event. `getPopupType()` returns `DROPDOWN_LIST` if the event is a result of the launch of the

LOV Search and Select dialog, and `SEARCH_DIALOG` if the event is the result of the user clicking the **Search** button in the dialog.

7. If you want users to be able to create a new item, create a popup dialog with the ID given in Step 5. See [Using Popup Dialogs, Menus, and Windows](#).
8. In the Components window, from the Operations panel, in the Behavior group, drag an **Auto Suggest Behavior** and drop it as child to the `inputComboboxListOfValues` component.

If you add auto suggest behavior, you must not set the `immediate` property to `true`. Setting `immediate` to `true` will cause validation to occur in the Apply Request Values phase and any validation errors may suppress the displaying of the suggestion list.

9. In the Properties window, for each of the auto-suggest attributes, enter the:
 - EL expression that resolves to the `suggestItems` method.

The method should return `List<javax.model.SelectItem>` of the `suggestItems`. The method signature should be of the form

```
List<javax.model.SelectItem>
suggestItems( javax.faces.context.FacesContext,
oracle.adf.view.rich.model.AutoSuggestUIHints)
```
 - EL expression that resolves to the `smartList` method. The method should return `List<javax.model.SelectItem>` of the smart list items.
 - Number of items to be displayed in the auto-suggest list. Enter -1 to display the complete list.

If you are implementing this method in a managed bean, the JSF page entry should have the format shown in the following example.

```
<af:inputComboboxListOfValues value="#{bean.value}" id="inputId">
  ...
  <af:autoSuggestBehavior
    suggestItems="#{bean.suggestItems}"
    smartList="#{bean.smartList}"
    maxSuggestedItems="7"/>
</af:inputComboboxListOfValues>
```

If the component is being used with a data model such as ADF Model, the `suggestItem` method should be provided by the default implementation.

10. If you are not using the component with ADF Model, create the `suggestItems` method to process and display the list:

```
List<javax.model.SelectItem> suggestItems( javax.faces.context.FacesContext,
oracle.adf.view.rich.model.AutoSuggestUIHints)
```

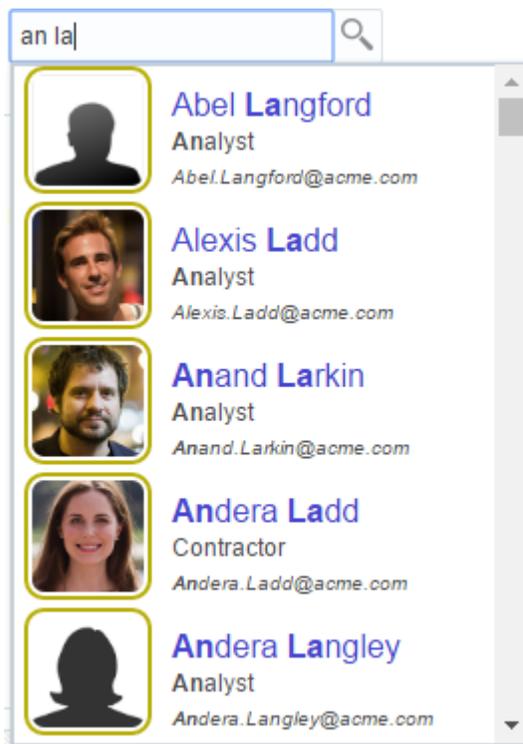
Using the InputSearch Component

Using the ADF `inputSearch` component you can filter from a list of values to search and highlight the matched suggestion. You must use the ADF `inputSearch` component with REST resource and `inputSearch` supports tag attributes to understand REST data.

The `inputSearch` component performs filtering and highlights the matched suggestions on the client. This component fetches the LOV list data from the REST resource call. For better performance, it is recommended to use

the `inputSearch` component with a REST resource that provides around 5000 rows. If the REST response returns more than 5000 rows or beyond the configured limit, the first 5000 rows are displayed. The `inputSearch` component performs filtering as specified on the component using the `filterAttributes` and the filter string is case-insensitive. This component performs search across all attributes that supports the search criteria `contains` and `startsWith`. By default, the display of suggestions is done using the standard template. You can customize the display of suggestions using clientside templates. [Figure 13-8](#) displays the sample standard template that shows the filtering across all attributes and highlights the matched suggestion.

Figure 13-8 Filters Across All Attributes and Matches Suggestions with Highlighting



The performance of the component is dependent on the appropriate caching strategy that you use while developing a web page. The `inputSearch` component uses REST services to fetch the suggestions. The caching mechanism that you can use is the default HTTP caching supported by browsers. You can select an appropriate caching strategy for the REST resource. The `inputSearch` component will be able to parse the response if the REST service adheres to the REST format. The REST response should be in JSON format. A sample REST response is given below, which was used to generate `inputSearch` list as displayed in [Figure 13-8](#), where the values of specified attributes display in the component.

```
[{
  "id": 2930,
  "dateOfBirth": "Jan 12, 1991",
  "deptName": "Accounting",
  "tags": "Excel, ",
  "email": "Abe.Chamberlain@acme.com",
  "hireDate": "May 16, 2004",
```

```

        "profileKey": "19_M.jpg",
        "lName": "Chamberlain",
        "deptLocation": "Bombay, India",
        "genderCode": "M",
        "fName": "Abe",
        "jobTitle": "Contractor"
    }, {
        "id": 4150,
        "dateOfBirth": "Aug 26, 1982",
        "deptName": "Customer Support",
        "tags": "negotiator, filing, ",
        "email": "Abe.Easter@acme.com",
        "hireDate": "Aug 23, 2000",
        "profileKey": "7_M.jpg",
        "lName": "Easter",
        "deptLocation": "New York, USA",
        "genderCode": "M",
        "fName": "Abe",
        "jobTitle": "Contractor"
    },
    ...

```

You can customize `inputSearch` by modifying the source code of the `.jspx` page. The following customizations are available for the `inputSearch` component.

- The `inputSearch` component understands the REST data with tag attributes. The data that is to be filtered and displayed should be specified in the tag attributes. You can specify the values from the REST response data for the tag attributes. See [What You May Need to Know About InputSearch Component Tags and REST Data](#).
- The `inputSearch` component fetches the LOV list data from the REST resource. Therefore, you must specify the REST resource for `af:inputSearch` component, which is the REST endpoint URL. You can use the `searchSection` tag to specify the REST resource path.

```

<af:inputSearch id="iSearch1"
...
    <af:searchSection dataUrl="/#{request.contextPath}/rest/employees" />
</af:inputSearch>

```

The `inputSearch` component does client filtering of the suggestions if the complete suggestions list is sent to the client on a simple request to the `dataUrl`. If the results set is paginated with the response having `hasMore` property set to true, then the component delegates the filtering to the REST service. For the server filtering, the `searchSection` supports an attribute named `filterParameters` that takes a JS callback name. The JS method is invoked while making the REST calls for server filtering and the callback returns the appropriate query parameters for the REST URL to perform filtering on the REST service. See [What You May Need to Know About SearchSection Attributes](#).

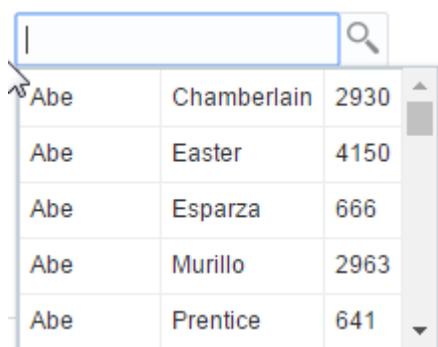
- By default, the `inputSearch` component suggestions are displayed in list format. You can customize the display of `inputSearch` suggestions either as a list or table. You can use the `contentMode` facet to customize the display.

```

<af:inputSearch contentMode="table/list" ... >
...

```

[Figure 13-9](#) and [Figure 13-10](#) display the default List and Table display modes.

Figure 13-9 Default List Display**Figure 13-10 Default Table Display**

- The `inputSearch` component supports clientside template to customize the presentation of suggestions. See [How to Customize InputSearch Component Display Modes](#).
- You can include other attributes to customize the display, such as short descriptions, selection converter, and auto submit for `inputSearch`. See [What You May Need to Know About InputSearch Attributes](#).
- You can build and configure a list of preferred suggestions and manage this suggestion list on a client by using the `af:suggestionSection` tag within the `af:inputSearch` component. By default, `af:suggestionSection` based list is shared across `inputSearch` components which are based on same REST URL. You can make them private to the `inputSearch` component by setting the `cacheKeyGenerator` attribute value to `privateList`. See [How to Set Attributes of Suggestion Section](#).

```
<af:inputSearch label="Label"
  valueAttribute="id" ...>
  <af:searchSection type="default" dataUrl="/rest/employees" />
  <af:suggestionSection />
</af:inputSearch>
```

When REST calls are made, the `inputSearch` component sets the `Accept` header to `application/json`. The JSON document should have a root property labeled `items` and its value should be an array. If an `items` property is not available, the first root property for which the value is an array is chosen as the list of suggestions. You can set the attributes in the `suggestionSection` to customize the

sharing and display of suggestion list. See [How to Set Attributes of Suggestion Section](#).

- The `inputSearch` component supports dependency-based filtering on the clientside. See [How to Specify Dependency-Based Filtering on InputSearch Components](#).

How to Use the InputSearch Component

You customize the `inputSearch` component by modifying the source code of the `.jspx` page. You can modify the tag attributes, modify the REST url, customize the display mode, or customize the suggestion section. You can also include clientside template suggestions and dependency-based filtering by modifying the source code. All the customizations are done by modifying the source code.

Before you begin:

- Ensure that the RESTful web service is running when running the `.jspx` page that contains the `af:inputSearch` component.
- See [Using the InputSearch Component](#) to understand the `inputSearch` component.

To customize the `inputSearch` to display data in tabular format:

1. In the `.jspx` page that contains the `af:inputSearch` LOV component, click the **Source** tab to view the source code.
2. Modify the `label` attribute to specify a label for the component.

```
label="Select a person from the list (Filter by name): "
```

See [Figure 13-11](#) to view the label that is displayed in the output.

3. Specify the value for `valueAttribute` (`id`), `displayAttributes` (`fName lName`), and `filterAttributes` (`fName lName jobTitle`) to filter by name and job title.

```
valueAttribute="id"
displayAttributes="fName lName"
filterAttributes="fName lName jobTitle"
```

See [What You May Need to Know About InputSearch Component Tags and REST Data](#).

4. Optionally, customize the display of the `inputSearch` component to display as a table by using the `contentMode` facet.

```
contentMode="table">
```

See [How to Customize InputSearch Component Display Modes](#).

5. Add a `searchSection` component and specify the REST endpoint URL in `searchSection`.

```
<af:inputSearch ...
  <af:searchSection type="default"
    dataUrl="/rest/employees?cache=etag&limit=5000" />
</af:inputSearch>
```

6. Optionally, include the `suggestionSection` tag to fetch the suggestion list using the REST data. The following code uses the `cacheKeyGenerator` attribute to return

the `clientId` of the `inputSearch` component to maintain the suggestion list as private on the `inputSearch` component.

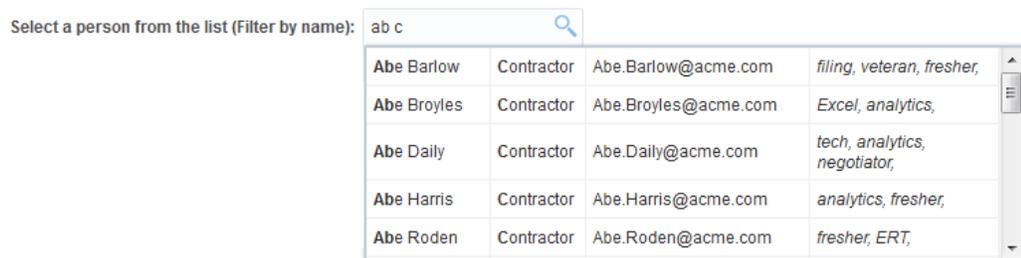
```
<af:suggestionsSection dontCache="tags1" displayCount="15"
cacheKeyGenerator="privateList" />
```

See [Configuring Suggestion List](#) and [How to Set Attributes of Suggestion Section](#).

The following code example shows the complete code as detailed in the above tasks, followed by the sample template as shown in [Figure 13-11](#).

```
<af:inputSearch id="iSearch1"
  label="Select a person from the list (Filter by name): "
  valueAttribute="id"
  displayAttributes="fName lName"
  filterAttributes="fName lName jobTitle tags"
  selectionConverter="#{inputSearchDemo.selectionConverter}"
  autoSubmit="true"
  contentMode="table">
  <af:validator validatorId="oracle.adfdemo.InputSearchDemoValidator" />
  <af:searchSection type="default"
    dataUrl="/#{request.contextPath}/rest/employees?
cache=etag&limit=5000" />
  <af:suggestionsSection dontCache="tags1" displayCount="15"
cacheKeyGenerator="privateList" />
  <f:facet name="contentStamp">
    <af:sanitized >
      <td>{{fName}} {{lName}}</td>
      <td>{{jobTitle}}</td>
      <td>{{email}}</td>
      <td><span style="font-style: italic;">{{tags}}</span></td>
    </af:sanitized>
  </f:facet>
</af:inputSearch>
```

Figure 13-11 Sample Output of Modified Code



What You May Need to Know About InputSearch Component Tags and REST Data

The `inputSearch` component supports three tag attributes to understand the REST data. The following example shows the sample tag attributes for the `inputSearch` component:

```
<af:inputSearch id="iSearch1"
  valueAttribute="id"
  displayAttributes="fName lName id"
```

```
        filterAttributes="fName lName jobTitle tags">  
</af:inputSearch>
```

- `valueAttribute` - specify a value that is unique on which the component is defined. In the REST response data sample shown above, the `id` attribute uniquely identifies the suggestion object and hence given as the input for `valueAttribute`.
- `displayAttributes` - specify a collection of item properties from the REST response data that is to be displayed in the input field on selection. The value to be displayed in the input field is a combination of `fName`, `lName`, and `id`. Therefore, these fields are specified in the `displayAttributes`. If you do not specify a value, this attribute takes the value of `valueAttribute`, by default.
- `filterAttributes` - specify a collection of item properties from the REST response data, based on which the data is filtered and the values displayed in the suggestions popup. The value to be displayed should be filtered on `fName`, `lName`, and `jobTitle`. Therefore, these fields are specified in the `filterAttributes`. If not specified, then it takes the value of `displayAttributes`.
The `inputSearch` component performs filtering and highlights the matched suggestions on the client as in [Figure 13-8](#).

What You May Need to Know About InputSearch Attributes

You can customize the filtering and display of suggestions by including attributes in `inputSearch` source code. [Table 13-2](#) provides the list of attributes that you can use in the `inputSearch` component.

Table 13-2 Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
<code>accessKey</code>	Char	Yes	<p>A character used to gain quick access to the form element specified by the <code>for</code> if set, or to this component itself, if it is a non-simple form element. If the same access key appears in multiple locations in the same page of output, the rendering user agent will cycle among the elements accessed by the similar keys.</p> <p>This attribute is sometimes referred to as the mnemonic.</p>
			<p> Note:</p> <p>The <code>accessKey</code> is triggered by browser-specific and platform-specific modifier keys. It even has browser-specific meaning. For example, Internet Explorer will set focus when you press <code>Alt+accessKey</code>. Firefox sets focus on some operating systems when you press <code>Alt+Shift+accessKey</code>. Firefox on other operating systems sets focus when you press <code>Ctrl+accessKey</code>. Refer your browser documentation for how it treats access keys.</p>
<code>attributeChangeListener</code>	<code>javax.el.MethodExpression</code>	Only EL	<p>A method reference to an attribute change listener. Attribute change events are not delivered for any programmatic change to a property. They are only delivered when a renderer changes a property without the application's specific request. An example of an attribute change events might include the width of a column that supported clientside resizing.</p>
<code>autoSubmit</code>	Boolean	Yes	<p>When set to <code>True</code> on a form element, the component will automatically submit when an appropriate action such as, a click, text change takes place. Partial submit like <code>autoSubmit</code> and any other components with <code>partialTriggers</code> pointing to this component is also submitted and re-rendered.</p> <p>The default value is <code>False</code>.</p>

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
binding	oracle.adf.view.rich.component.rich.input.RichInputSearch	Only EL	An EL reference that will store the component instance on a bean. This can be used to give programmatic access to a component from a backing bean, or to move creation of the component to a backing bean.
changed	Boolean	Yes	When set to true, the changed indicator icon will be displayed on the component. The default value is <code>False</code> .
changedDesc	String	Yes	The text commonly used by user agents to display tool tip text on the changed indicator icon. Default value is <code>Changed</code> . The behavior of the tool tip is controlled by the user agent. For example, Firefox 2 truncates long tool tips.
clientComponent	Boolean	Yes	Specifies if a clientside component is generated or not. A component may be generated whether or not this flag is set, but if client Javascript requires the component object, this must be set to true to guarantee the component's presence. Client component objects that are generated today, by default, may not be present in the future. Setting this flag is the only way to guarantee a component's presence, and clients cannot rely on implicit behavior. However, there may be an impact on performance when you set this flag, therefore, you should avoid turning on client components unless absolutely necessary. For the components <code>outputText</code> and <code>outputFormatted</code> , setting the <code>clientComponent</code> to true will render id attribute for the HTML document object model. This ID attribute can alternatively be generated by setting <code>oracle.adf.view.rich.SUPPRESS_IDS</code> to auto in <code>web.xml</code> . The default value is <code>False</code> .
columns	int	Yes	The size of the text control specified by the number of characters shown. The number of columns is estimated based on the default font size of the browser.
contentMode	oracle.adf.view.rich.component.rich.input.RichInputSearch.ContentMode	Yes	Indicates the rendering mode of <code>contentStamp</code> facet. Valid values are <code>list</code> and <code>table</code> . For programmatic usage, use the enumerated constants from <code>RichInputSearch.ContentMode</code> . The default value is <code>List</code> .

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
contentStyle	String	Yes	The style of the content piece of the component. You can style width by setting this attribute like width: 100px. Because of browser CSS precedence rules, CSS rendered on a DOM element takes precedence over external style sheets like the skin file. Therefore skins will not be able to override what you set on this attribute.
converter	javax.faces.convert.Converter	Yes	A converter object
criteria	oracle.adf.view.rich.component.rich.input.RichInputSearch.Criteria	Yes	Indicates the search criteria. Valid Values are startsWith, contains, and auto. For programmatic usage, use the enumerated constants from RichInputSearch.Criteria. that displays startsWith matches before displaying contains matches in the suggestions panel. The default value is auto, which takes contains search criteria.
customizationId	String	Yes	This attribute is deprecated. The id attribute should be used when applying persistent customization. This attribute will be removed in the next release.
disabled	Boolean	Yes	Specifies whether the element is enabled or disabled. Unlike a readOnly component, a disabled component is unable to receive focus. If the component has the potential to have a scrollbar, and you want the user to be able to scroll through the component's text or values, use the readOnly attribute, not the disabled attribute. The default value is False.
displayAttributes	java.util.List	Yes	The collection item properties from the REST response data whose values represent the selection. This attribute decides the value to be displayed in the input field on selection. If displayAttributes is not specified, it defaults to valueAttribute.
editable	String	Yes	The editable look and feel to use for input components. You can use the following values: <ul style="list-style-type: none"> always - the input component always look editable onAccess - the input will only look editable when accessed (hover, focus) inherit - indicates to use the component parent's setting. None of the ancestor components define always or onAccess, then inherit will be used, which is the default value.

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
filter	String	Yes	<p>This attribute holds the name of the JS callback that would perform <code>Array.filter</code> operation on the collection returned from the REST response. The component instance will be set as context during JS callback invocation. The row object passed to the callback will have the structure below:</p> <ul style="list-style-type: none"> • <code>data</code>: The raw row data • <code>index</code>: The index for the row • <code>key</code>: The key value for the row <p>See Dependency Based Filtering.</p>
filterAttributes	String	Yes	<p>The collection item properties from the REST response data to be used for filtering suggestions. If <code>filterAttributes</code> is not specified, it defaults to <code>displayAttributes</code>. Where <code>contentStamp</code> facet is not provided, the values displayed in the suggestions popup is also decided by this attribute.</p>
helpTopicId	String	Yes	<p>The id used to look up a topic in a <code>helpProvider</code>. Help information should be provided using the <code>helpTopicId</code>.</p> <p>It is preferred to use <code>helpTopicId</code> instead of <code>shortDesc</code> to provide help information.</p>

 **Note:**

If you specify an attribute that is not displayed in the suggestions panel as a filterable attribute, it could be confusing for the end user to see the suggestions being filtered based on such an attribute.

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
id	String	No	<p>The identifier for the component. Every component may be named by a component identifier that must conform to the following rules:</p> <ul style="list-style-type: none"> • They must start with a letter as defined by the <code>Character.isLetter()</code> method or underscore (<code>_</code>). • Subsequent characters must be letters as defined by the <code>Character.isLetter()</code> method, digits as defined by the <code>Character.isDigit()</code> method, dashes (<code>-</code>), or underscores (<code>_</code>). To minimize the size of responses generated by JavaServer Faces, it is recommended that component identifiers be as short as possible. If a component has been given an identifier, it must be unique in the namespace of the closest ancestor to that component that is a <code>NamingContainer</code>.
immediate	Boolean	Yes	<p>Specifies whether the value is converted and validated immediately in the Apply Request Values phase, or is handled in the Process Validators phase. By default, the values are converted and validated together in the Process Validators phase. However, if you need access to the value of a component during Apply Request Values, for example, if you need to get the value from an <code>actionListener</code> on an immediate command button, then set this to <code>immediate</code>.</p> <p>The default value is <code>False</code>.</p>
inlineStyle	String	Yes	<p>The CSS styles to use for this component and intended for basic style changes. The <code>inlineStyle</code> is a set of CSS styles that are applied to the root DOM element of the component. Because of browser CSS precedence rules, CSS rendered on a DOM element takes precedence over external style sheets like the skin file. Therefore, skins will not be able to override what you set on this attribute.</p> <p>If the <code>inlineStyle</code> CSS properties do not affect the DOM element, then create a skin and use the skinning keys like <code>::label</code> or <code>::icon-style</code> to target a particular DOM elements.</p>

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
label	String	Yes	<p>The label of the component. If you want the label to appear above the control, use a <code>panelFormLayout</code>.</p> <p>For all input components and compound components that have an input component part for which a label can be assigned, the component requires a label. This can be accomplished in one of the following ways:</p> <ul style="list-style-type: none"> • a <code>label</code> value on the component • a <code>labelAndAccessKey</code> value on the component • wrapping the component with an <code>af:panelLabelAndMessage</code> component that has a <code>label</code> or <code>labelAndAccessKey</code> value and a <code>for</code> value specifying the component id • having a corresponding <code>af:outputLabel</code> component that has a <code>value</code> or <code>valueAndAccessKey</code> value and a <code>for</code> value specifying the component id
			<p>All methods may not be available for all the components.</p>
labelAndAccessKey	String	Yes	An attribute that will simultaneously set both the <code>label</code> and <code>accessKey</code> attributes from a single value, using conventional ampersand notation.
labelStyle	String	Yes	The CSS styles to use for the label of this component. The <code>labelStyle</code> is a set of CSS styles that are applied to the label DOM element of the component. This allows a label to be styled without requiring a new skin definition.

 **Note:**

Any one of these approaches is sufficient, though `label` or `labelAndAccessKey` is the most convenient and should be used where possible

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
<code>partialTriggers</code>	<code>String[]</code>	Yes	<p>The IDs of the components that should trigger a partial update. This component listens on the trigger components. If one of the trigger components receives an event that causes it to update in some way, this component will request to be updated too. Identifiers are relative to the source component, and must account for <code>NamingContainers</code>. If your component is already inside of a naming container, you can use a single colon to start the search from the root of the page, or multiple colons to move up through the <code>NamingContainers</code>:</p> <ul style="list-style-type: none"> <code>:</code> starts the search from root of the page <code>::</code> comes out of the component's naming container and begins the search from there <code>:::</code> comes out of two naming containers and begins the search from there
<code>placeholder</code>	<code>String</code>	Yes	<p>Text to be displayed in the <code>input</code> component when a value is not present.</p> <p>Placeholder text is meant to provide an example, and does not act as a label for the field. Components that have placeholders set still require labels.</p>
<code>protectionKey</code>	<code>String</code>	Yes	Protection key for this component.
<code>readOnly</code>	<code>Boolean</code>	Yes	<p>Specifies whether the control is displayed as an editable field or as an output-style text control. Unlike a disabled component, a <code>readOnly</code> component is able to receive focus.</p> <p>The default value is <code>False</code>.</p>
<code>rendered</code>	<code>Boolean</code>	Yes	<p>Specifies if the component is rendered. When set to false, no output will be delivered for this component. The component is not rendered any way and is not visible on the client.</p> <p>If you want to change a component's rendered attribute from false to true using partial page rendering, set the <code>partialTrigger</code> attribute of its parent component so the parent refreshes and in turn will render this component.</p> <p>The default value is <code>True</code>.</p>
<code>required</code>	<code>Boolean</code>	Yes	<p>Specifies if a non-null, non-empty value must be entered. If false, required validation logic is not executed when the value is null or empty.</p> <p>The default value is <code>False</code>.</p>
<code>requiredMessageDetail</code>	<code>String</code>	Yes	Specifies the message to be displayed, if required validation fails.

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
rowCount	int	Yes	<p>Max number of rows shown in the suggestion panel before scrolling. If -1 is the value, the row count is determined by the ADF skin property <code>-tr-row-count</code>.</p> <p>The default value is -1.</p>
selectionConverter	javax.el.MethodExpression	Only EL	<p>Specifies a method reference to convert the members of the Map object returned by <code>getSelection()</code> method. The method should convert both the key and value returned by <code>getSelection()</code> method to the appropriate Java types.</p>
shortDesc	String	Yes	<p>Specifies the short description of the component. The <code>shortDesc</code> text may be used in two different ways, depending on the component.</p> <p>For components with images, the <code>shortDesc</code> is often used to render an HTML <code>alt</code> attribute for the image. Refer <code>helpTopicId</code> for more information.</p> <p><code>shortDesc</code> is also commonly used to render an HTML <code>title</code> attribute, which is used by user agents to display tooltip help text.</p> <p>The behavior for the <code>tooltip</code> is controlled by the user agent. For example, Firefox 2 truncates long tooltips.</p> <p>For form components, the <code>shortDesc</code> is displayed in a note window.</p> <p>For components that support the <code>helpTopicId</code> attribute and are not using the <code>shortDesc</code> as image alt text, it is recommended that you use <code>helpTopicId</code> instead of <code>shortDesc</code> as it is more flexible and provides more accessible descriptive text than the use of the <code>title</code> attribute.</p>
showRequired	Boolean	Yes	<p>Specifies if the associated control displays a visual indication of required user input. If a required attribute is also present, both the required attribute and the <code>showRequired</code> attribute must be false so that the visual indication is not displayed.</p> <p>If you have a field that is initially empty and is required only if some other field on the page is touched, then you can use the <code>showRequired</code> attribute.</p> <p>The default value is <code>False</code>.</p>

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
simple	Boolean	Yes	<p>Controls if a component provides label support or not. If the value is set to True, the component does not display the label. The <code>label</code>, <code>labelAndAccessKey</code>, <code>accessKey</code>, <code>showRequired</code>, and <code>help</code> facets are ignored and displays a simpler layout.</p> <p>You can use this attribute when you use a component repeatedly used as in table, where a label is not required.</p> <p>The default value is <code>False</code>.</p>
styleClass	String	Yes	<p>Specifies a CSS style class to use for this component. The style class can be defined in a jsp page or in a skinning CSS file. For example you can use one of the public style classes, <code>AFInstructionText</code>.</p>
unsecure	java.util.Set	Yes	<p>A whitespace separated list of attributes whose values can be set only on the server, but should be able to set on the client. Currently, this is supported only for the disabled attribute.</p>
			<div style="border: 1px solid #0070C0; padding: 10px; background-color: #E6F2FF;"> <p> Note:</p> <p>To set a property on the client, you must use the <code>setProperty('attribute', newValue)</code> method, and not the <code>setXXXAttribute(newValue)</code> method.</p> </div>
usage	String	Yes	<p>For example, if you have unsecure=disabled, then on the client you can use the <code>setProperty('disabled', false)</code>, method. The <code>setDisabled(false)</code> does not work and provides a javascript error that <code>setDisabled</code> is not a function.</p> <p>The usage attribute will set the type of the input to allow for different html types, such as search. The valid values are <code>auto</code>, <code>text</code>, and <code>search</code>.</p> <p>The default value is <code>auto</code> which replicates text.</p>
validator	javax.faces.el.MethodBinding	Only EL	<p>Specifies a method reference to a validator method.</p>
value	Object	Yes	<p>Specifies the value of the component. If the EL binding for the value, points to a bean property with a getter but no setter, and if this is an editable component, then the component is rendered in a read-only mode.</p>

Table 13-2 (Cont.) Attributes Supported by inputSearch Component

Name	Type	Supports EL	Description
valueAttribute	String	Yes	Specifies the collection item property in REST response data on which the component is defined on. The values of this attribute should be unique. For example, if the component is used for displaying and filtering employee details, Employee Number is a good candidate to be chosen as value attribute.
valueChangeListener	javax.faces.el.MethodBinding	Only EL	Specifies a method reference to a value change listener.
visible	Boolean	Yes	Specifies the visibility of the component. If set to false, the component is hidden on the client. Unlike the <code>rendered</code> attribute, this does not affect the lifecycle on the server. The component may have its bindings executed, and the visibility of the component can be toggled on and off on the client, or toggled with PPR. When <code>rendered</code> is false, the component is not any way be rendered, and cannot be made visible on the client. In most cases, use the <code>rendered</code> property instead of the <code>visible</code> property. This attribute is not supported on the following renderkits: <code>org.apache.myfaces.trinidad.core</code> . The default value is <code>True</code> .

What You May Need to Know About SearchSection Attributes

The `af:searchSection` component is basically used to specify the REST endpoint URL. The `af:searchSection` supports other attributes that you can use to customize the search section, such as search type, search id and so on. The following example shows the code sample of `af:searchSection`:

```
<af:inputSearch label="Label" valueAttribute="id">
  <af:searchSection type="default"
    dataUrl="/rest/employees" />
</af:inputSearch>
```

[Table 13-3](#) lists the attributes that you can use in the `af:searchSection`.

Table 13-3 Attributes that You can Use with searchSection

Attribute Name	Type	Supports EL?	Description
attributeChangeListener	javax.el.MethodExpression	Only EL	A method reference to an attribute change listener. Attribute change events are not delivered for any programmatic change to a property. They are only delivered when a renderer changes a property without the application's specific request. An example of an attribute change events might include the width of a column that supported clientside resizing.
binding	oracle.adf.view.rich.component.rich.data.RichSearchSection	Only EL	An EL reference that will store the component instance on a bean. This can be used to give programmatic access to a component from a backing bean, or to move creation of the component to a backing bean.
dataUrl	String	Yes	<p>REST service endpoint URI that provides a list of suggestions for the search section. This attribute supports the following types of URIs:</p> <p>absolute - an absolute path. For example, <code>http://samplerest.com/rest/Emp</code></p> <p>context relative - a path relatively based on the web application's context root. For example, <code>/rest/Emp</code></p> <p>server relative - a path relatively based on the web server by application name. For example, <code>//RESTWebService-context-root/rest/Emp</code></p> <p>The REST response should strictly be in JSON format. The component will set the Accept header to <code>application/json</code> for the REST calls made.</p> <p>The JSON document should have a root property labeled <code>items</code> and its value should be an array. If <code>items</code> property is not available, the first root property whose value is an array is chosen as the list of suggestions.</p>

 **Note:**

If the REST endpoint is on a different origin, CORS needs to be setup on the endpoint.

Table 13-3 (Cont.) Attributes that You can Use with searchSection

Attribute Name	Type	Supports EL?	Description
filterParameters	String	YES	Attribute to set the parameters relevant for filtering of the suggestions on the REST service. Server filtering is attempted when the collection size is beyond the configured fetch limit set on the REST service and the REST response has hasMore property set. Otherwise the filtering is completely done on the client and this attribute is not invoked.

 **Note:**

When filtering is done on the server side, the dependency based filtering also happens on the server and not on the client.

When the master collection size is beyond the configured fetch limit, a converter needs to be attached to the `inputSearch` component to derive the display value from the `displayAttributes`.

This attribute holds the name of the JS callback and JS callback is invoked when a REST call is being made in response to user action of filtering. The `searchSection` client component instance will be set as context during JS callback invocation. A request and context objects will be passed as parameters to the function. The callback returns the configured passed in request object.

The request object has the following structure:

- **queryString:** The string set on this property will be appended as a query parameter to the REST URL while making the HTTP request call. If the **queryString** has a query parameter that is already present in the base URL, such a query parameter on the base URL will be replaced with the new value in **queryString**.
- **method:** The request method. For example, GET, POST. Default is GET
- **body:** The request body for POST requests

The context object has the following structure:

- **criteria:** `AdfRichInputSearch.CRITERIA_STARTS_WITH` or `AdfRichInputSearch.CRITERIA_CONTAINS`, the search that needs to be performed
- **searchTerms:** Array of search terms
- **filterAttributes:** Array of attributes across which the search needs to be filtered

Table 13-3 (Cont.) Attributes that You can Use with searchSection

Attribute Name	Type	Supports EL?	Description
id	String	No	The identifier for the component. Every component may be named by a component identifier that must conform to the following rules: They must start with a letter (as defined by the <code>Character.isLetter()</code> method) or underscore (<code>_</code>). Subsequent characters must be letters (as defined by the <code>Character.isLetter()</code> method), digits as defined by the <code>Character.isDigit()</code> method, dashes (<code>-</code>), or underscores (<code>_</code>). To minimize the size of responses generated by Java Server Faces, it is recommended that component identifiers be as short as possible. If a component has been given an identifier, it must be unique in the namespace of the closest ancestor to that component that is a <code>NamingContainer</code> (if any).
rendered	Boolean	Yes	The value is set to <code>True</code> , by default. This attribute specifies if the component is rendered or not. When set to <code>false</code> , no output will be delivered for this component (the component will not in any way be rendered, and cannot be made visible on the client). If you want to change a component's rendered attribute from <code>false</code> to <code>true</code> using partial page rendering, set the <code>partialTrigger</code> attribute of its parent component so the parent refreshes and in turn will render this component.
type	String	Yes	The type of the search section. Currently, only default <code>searchSection</code> is supported.

How to Customize InputSearch Component Display Modes

You can customize the display of suggestions on the client by using the client template feature supported by the `inputSearch` component. The rendering of the component and submission of value happens through the JSF lifecycle. The template system supported by the component is called Mustache, a web template system that may be embedded into HTML and relies on a template engine to expand the template at runtime. See [Mustache Manual](#). You can include the value from the REST response as Mustache tags within the `af:sanitized` tag of `jspx` page. You can use the `contentStamp` and `contentHeader` facets in the `.jspx` page source code to provide the template. Mustache tags are evaluated for each suggestion object and the expanded template is rendered as a suggestion item in the suggestions panel.

Basically, the property names in the JavaScript suggestion object are to be referenced as mustache tags in the template. The component would evaluate the template replacing the mustache tags with the values in the suggestion object. Depending on the `contentMode` attribute, the component evaluates the mustache expressions and dumps it as is in the HTML either inside a `` tag or a `<tr>` tag to display the suggestion items.

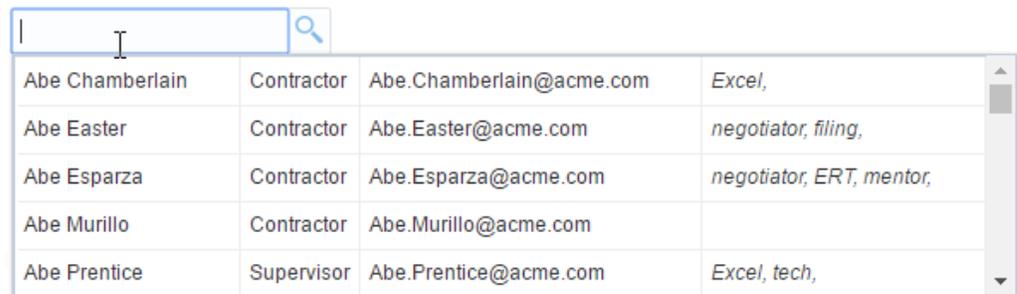
The following example shows the sample code for tabular display without any header, followed by the sample template as shown in [Figure 13-12](#).

```

<af:inputSearch contentMode="table" ... >
  ...
  <f:facet name="contentStamp">
    <af:sanitized >
      <td>{{fName}} {{lName}}</td>
      <td>{{jobTitle}}</td>
      <td>{{email}}</td>
      <td><span style="font-style: italic;">{{tags}}</span></td>
    </af:sanitized>
  </f:facet>
</af:inputSearch>

```

Figure 13-12 Sample Tabular Display



The screenshot shows a search bar at the top with a magnifying glass icon. Below it is a table with five rows of employee data. The table has four columns: Name, Job Title, Contact Email, and Specializations. The data is as follows:

Abe Chamberlain	Contractor	Abe.Chamberlain@acme.com	<i>Excel,</i>
Abe Easter	Contractor	Abe.Easter@acme.com	<i>negotiator, filing,</i>
Abe Esparza	Contractor	Abe.Esparza@acme.com	<i>negotiator, ERT, mentor,</i>
Abe Murillo	Contractor	Abe.Murillo@acme.com	
Abe Prentice	Supervisor	Abe.Prentice@acme.com	<i>Excel, tech,</i>

The following example shows the sample code for tabular display with `contentHeader`, followed by the sample template as shown in [Figure 13-13](#).

```

<af:inputSearch contentMode="table" ... >
  ...
  <f:facet name="contentHeader">
    <af:sanitized >
      <th>Name</th>
      <th>Title</th>
      <th>Contact Email</th>
      <th><span style="font-style: italic;">Specializations</span></th>
    </af:sanitized>
  </f:facet>
  <f:facet name="contentStamp">
    <af:sanitized >
      <td>{{fName}} {{lName}}</td>
      <td>{{jobTitle}}</td>
      <td>{{email}}</td>
      <td><span style="font-style: italic;">{{tags}}</span></td>
    </af:sanitized>
  </f:facet>
</af:inputSearch>

```

Figure 13-13 Sample Tabular Display with Header Row

Name	Title	Contact Email	Specializations
Abe Chamberlain	Contractor	Abe.Chamberlain@acme.com	<i>Excel,</i>
Abe Easter	Contractor	Abe.Easter@acme.com	<i>negotiator, filing,</i>
Abe Esparza	Contractor	Abe.Esparza@acme.com	<i>negotiator, ERT, mentor,</i>
Abe Murillo	Contractor	Abe.Murillo@acme.com	
Abe Prentice	Supervisor	Abe.Prentice@acme.com	<i>Excel, tech,</i>

The following example shows the sample code for list display, followed by the sample template as shown in [Figure 13-14](#).

```
<af:inputSearch ... >
  ...
  <f:facet name="contentStamp">
    <af:sanitized >
      <span style="font-size:1.2em">{{fName}} {{lName}}</span> &nbsp;
      <span style="font-size:0.9em; padding-left: 8px">{{jobTitle}}</span>
      <br/>
      <b>Tags: </b><span style="font-style: italic;">{{tags}}</span>
    </af:sanitized>
  </f:facet>
</af:inputSearch>
```

Figure 13-14 Sample List Display

Abe Chamberlain	Contractor	<i>Tags: Excel,</i>
Abe Easter	Contractor	<i>Tags: negotiator, filing,</i>
Abe Esparza	Contractor	<i>Tags: negotiator, ERT, mentor,</i>
Abe Murillo	Contractor	<i>Tags:</i>
Abe Prentice	Supervisor	<i>Tags: Excel, tech,</i>

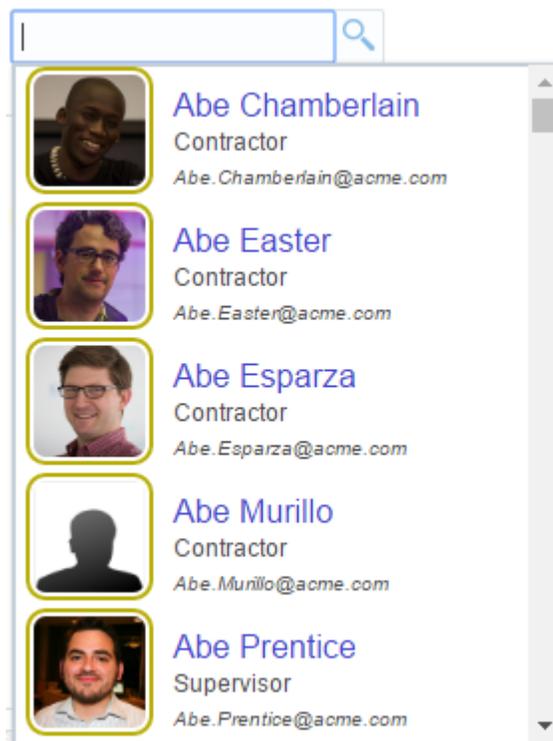
The following example shows the sample code for advanced list template, followed by the sample template as shown in [Figure 13-15](#).

```

<af:inputSearch ... >
  ...
  <f:facet name="contentStamp">
    <af:sanitized>
      <div style="height: 68px; line-height: 18px; white-space: nowrap">
        <div style="padding: 0px; margin: 0px; width: 64px; display: inline-block;">
          
        </div>
        <div style="padding-left: 10px; display: inline-block;">
          <div style="color: #4646D0;font-size: medium;">{{fName}} {{lName}}</div>
          <div>{{jobTitle}}</div>
          <div style="font-size: smaller;font-style: italic;">{{email}}</div>
        </div>
      </div>
    </af:sanitized>
  </f:facet>
</af:inputSearch>

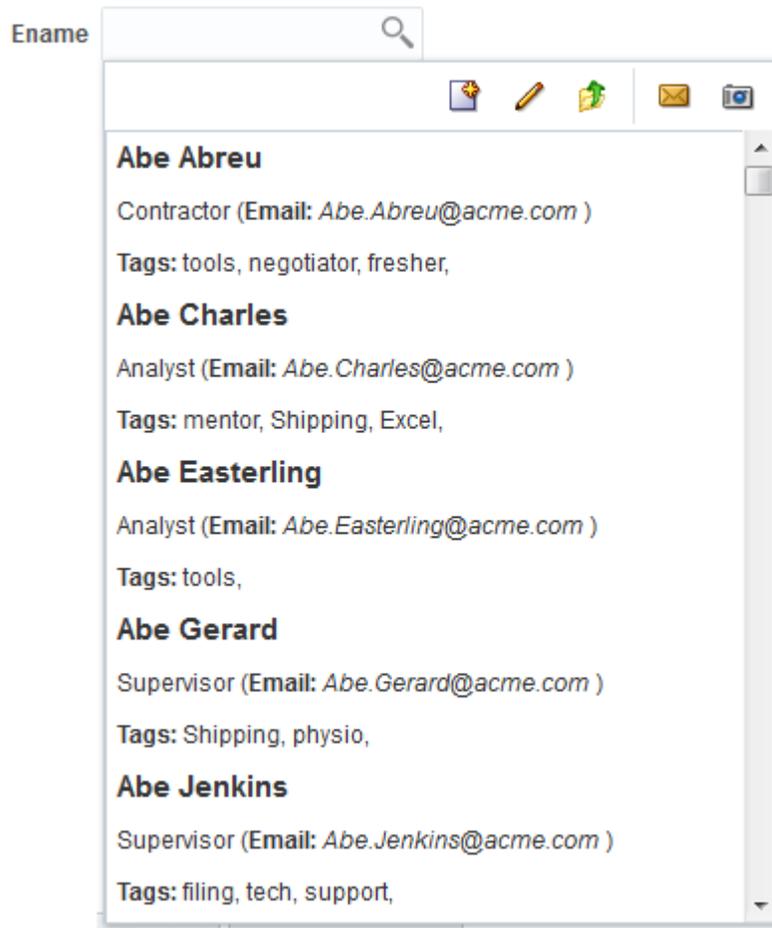
```

Figure 13-15 Sample Advanced List Display



How to Add Custom Buttons to Suggestions Popup

The `inputSearch` component includes a `facet toolbar` that renders a section in the suggestions panel, which you can use to place custom controls such as tool bar items of your choice. [Figure 13-16](#) displays the suggestion panel.

Figure 13-16 Suggestions Panel Displaying the Toolbar Facet

The following example shows the sample code for including the toolbar facet.

```
<af:searchSection type="default" dataUrl="#{inputSearchDemo.dataUrl}"
partialTriggers="ctb3" />
  <f:facet name="toolbar">
    <af:toolbar id="pgl14">
      <af:group id="g1">
        <af:commandToolBarButton shortDesc="Create" icon="/images/
new_ena.png" id="ctb1">
          <af:showPopupBehavior popupId="createpopup"/>
        </af:commandToolBarButton>
        <af:commandToolBarButton icon="/images/update_ena.png"
shortDesc="Update the searchSection URL on client" id="ctb2">
          <af:clientListener type="action" method="updateSuggestionCount" />
        </af:commandToolBarButton>
        <af:commandToolBarButton icon="/images/uplevel.gif"
actionListener="#{inputSearchDemo.updateDataUrl}"
shortDesc="Update the searchSection URL on server" id="ctb3"/>
      </af:group>
      <af:group id="g2">
        <af:commandToolBarButton shortDesc="E-mail" icon="/images/
email.gif" id="ctb4"/>
        <af:commandToolBarButton shortDesc="Snapshot" icon="/images/
snapshot.gif" id="ctb5"/>
      </af:group>
    </af:toolbar>
  </f:facet>
</af:searchSection>
```

```

        </af:group>
    </af:toolbar>
</f:facet>

```

What You May Need to Know About SuggestionSection Component

The `suggestionSection` component creates and manages the most recently used (MRU) suggestion lists based on the frequency and recentness of the suggestions used. The preferred list of suggestion items is stored in the browser's storage object. The `suggestionSection` component manages a list of preferred suggestions and gets displayed based on the various factors on the client:

- The parent `inputSearch` component retrieves the list managed by the `suggestionSection` and displays the list in the UI.
- The `inputSearch` component might not display the preferred suggestion list managed by `af:suggestionSection`, if the master list is small.
- The length of the master list is determined the first time the suggestion popup is launched. Therefore, if you launch the suggestion popup without much suggestions the first time, the `inputSearch` will still display the preferred suggestion list even if the length of the suggestion list is small.
- The list managed by the `suggestionSection` component is a subset of the suggestions retrieved via the REST service.

The following example shows the `suggestionSection` tag within the `inputSearch` component:

```

<af:inputSearch label="Label" valueAttribute="id" ...>
    <af:searchSection type="default" dataUrl="/rest/employees" />
    <af:suggestionsSection />
</af:inputSearch>

```

How to Set Attributes of Suggestion Section

By default, `af:suggestionsSection` based list is shared across `inputSearch` components which are based on same REST URL. You can use the following attributes in the `suggestionSection` to customize the sharing and display of suggestion list.

- `cacheKeyGenerator` Use this attribute to keep the suggestion list as private per component or to remove the cache search parameter from the URL to be used as `localStorage` key. Setting this attribute manages a list of suggestions completely on the client.

```

<af:inputSearch ...>
    ...
    <af:suggestionsSection cacheKeyGenerator="ignoreCacheAttr"/"privateList" />
</af:inputSearch>

```

- `displayCount` Use this attribute to customize the number of items that are to be displayed in the preferred list.

```

<af:inputSearch ...>
    ...
    <af:suggestionsSection displayCount="15" />
</af:inputSearch>

```

- `dontCache` Use this attribute to not to store the specified fields in the browser's `localStorage` for the REST data.

```
<af:inputSearch ...>
...
  <af:suggestionsSection dontCache="lName jobTitle" />
</af:inputSearch>
```

In this code sample, the `lName` and `jobTitle` attributes from the suggestion items retrieved from the REST call will not be stored in the `localStorage`.

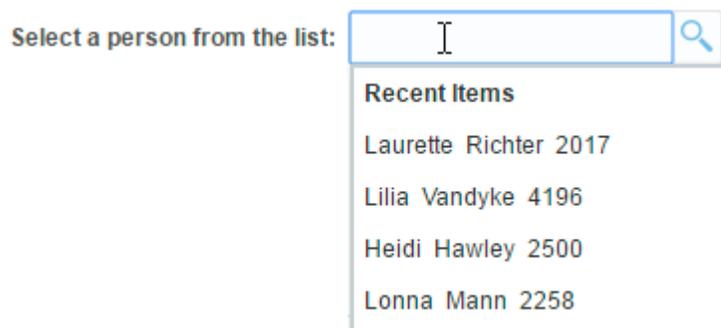
See [What You May Need to Know About suggestionSection Attributes](#) for other attributes that you can use with `suggestionSection` tag.

You can further customize the suggestion list by including a header facet. The `suggestionSection` supports only header facet that allows the child components within `af:sanitized`. The following example shows the sample code for the header facet that includes **Recent Items** as header in the suggestion list.

```
<af:inputSearch id="iSearch4"
  label="Select a person from the list (Filter by name and job title): "
  valueAttribute="id"
  displayAttributes="fName lName"
  filterAttributes="fName lName jobTitle"
  selectionConverter="{inputSearchDemo.selectionConverter}">
  <af:searchSection type="default" dataUrl="/rest/employees?
cache=etag&limit=5000" />
  <af:suggestionsSection>
    <f:facet name="header">
      <af:sanitized>
        <b>Recent Items</b> <!-- The "Recent Items" text in the screenshot below is
because of this line -->
      </af:sanitized>
    </f:facet>
  </af:suggestionsSection>
</af:inputSearch>
```

Figure 13-17 shows the output for the above code.

Figure 13-17 Output of the header Facet in `suggestionSection`



What You May Need to Know About suggestionSection Attributes

Table 13-4 lists the attributes that you can use in the `af:suggestionSection`.

Table 13-4 Attributes Used with suggestionSection

Attribute Name	Type	Supports EL?	Description
attributeChangeListener	javax.el.MethodExpression	Only EL	A method reference to an attribute change listener. Attribute change events are not delivered for any programmatic change to a property. They are only delivered when a renderer changes a property without the application's specific request. An example of an attribute change events might include the width of a column that supported client-side resizing.
binding	oracle.adf.view.rich.component.rich.data.RichSuggestionSection	Only EL	An EL reference that will store the component instance on a bean. This can be used to give programmatic access to a component from a backing bean, or to move creation of the component to a backing bean.
cacheKeyGenerator	String	Yes	The attribute holds the name of a JS function. The return value of this function will be used as the key for client storage. The normalized URL of the default searchSection is passed as a parameter to the function, and the function's "this" keyword refers to the parent inputSearch component instance. Sample JS method: <pre>function privateList(normalizedUrl) { // "this" refers to parent inputSearch component return this.getClientId(); }</pre>
display count	int	Yes	Maximum number of suggestions to be displayed in the UI. The default value is 5.
dontCacheSet	java.util.Set	Yes	A set of attribute names, whose values from the row data should not be cached on the client.
id	String	No	The identifier for the component. Every component may be named by a component identifier that must conform to the following rules: They must start with a letter (as defined by the Character.isLetter() method) or underscore (_). Subsequent characters must be letters (as defined by the Character.isLetter() method), digits as defined by the Character.isDigit() method, dashes (-), or underscores (_). To minimize the size of responses generated by Java Server Faces, it is recommended that component identifiers be as short as possible. If a component has been given an identifier, it must be unique in the namespace of the closest ancestor to that component that is a NamingContainer (if any).

Table 13-4 (Cont.) Attributes Used with suggestionSection

Attribute Name	Type	Supports EL?	Description
rendered	Boolean	Yes	The value is set to True, by default. This attribute specifies if the component is rendered or not. When set to false, no output will be delivered for this component (the component will not in any way be rendered, and cannot be made visible on the client). If you want to change a component's rendered attribute from false to true using partial page rendering, set the partialTrigger attribute of its parent component so the parent refreshes and in turn will render this component.

How to Specify Dependency-Based Filtering on InputSearch Components

When you use `filterAttributes` in the `inputSearch` component the REST response data is filtered and the values are displayed in the suggestions popup. The `inputSearch` component further supports dependency-based filtering on clientside. To further filter the suggestion list on the clientside, you can use the `filter` attribute that holds the name of the JavaScript callback that would perform `Array.filter` operation on the collection returned from the REST response. The component instance will be set as context during JavaScript callback invocation.

The row object passed to the callback will have the structure below:

- `data`: The raw row data
- `index`: The index for the row
- `key`: The key value for the row

The following example shows the sample code snippet for dependency based filtering that lists the Employee suggestion list based on the Department value, followed by the sample template as shown in [Figure 13-18](#).

```
<af:inputSearch label="Employee" filter="filterEmpForDept" ...>
  <af:suggestionSection type="default" dataUrl="/rest/employees" />
</af:inputSearch>
```

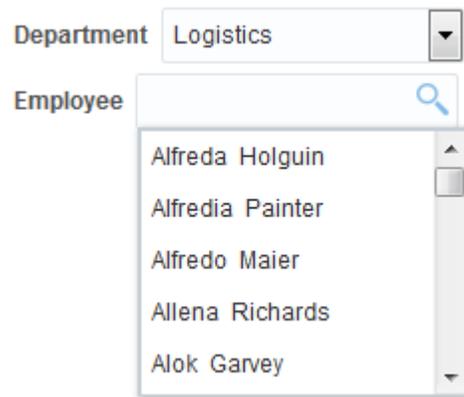
Sample JS:

```
var filterValue;
var filterKey = "deptName";
function filterEmpForDept(rowObj, collectionIndex)
{
  // var component = this;
  if (collectionIndex == 0)
  {
    // get the filterValue once for the zeroth row
    // and use the same for subsequent row filtering
    var parentComp = AdfPage.PAGE.findComponent
      ('parent component id whose value will determine the suggestions to be displayed
in inputSearch');
    var parentCompValue = parentComp.getValue();
    filterValue = parentCompValue ? parentComp.getSelectItems()
```

```
[parentComp.getValue()].getLabel() : null;  
}  
  
if (filterValue && rowObj.data[filterKey] == filterValue)  
    return true;  
return false;  
}
```

Figure 13-18 shows different Employee suggestion list based on the Department value.

Figure 13-18 Dependency-Based Filtering in inputSearch Component



14

Using Query Components

This chapter describes how to use the `query` and `quickQuery` search panel components. It describes how to configure the `query` component with search criteria, how to add and delete search criteria dynamically, and how to create and personalized saved searches. It also describe how to configure and add the `quickQuery` component to the page.

This chapter includes the following sections:

- [About Query Components](#)
- [Creating the Query Data Model](#)
- [Using the quickQuery Component](#)
- [Using the query Component](#)

About Query Components

ADF Faces provides query components that allows implementation of search functionality in your application. Users can personalize searches, save search, and can perform a `query` or a `quickQuery` against a search criteria.

The `query` and `quickQuery` components are used to search through data sets. The `query` component provides a comprehensive set of search criteria and controls, while the `quickQuery` component can be used for searching on a single criterion.

The `query` component supports the following functionality:

- Selecting and searching against multiple search criteria
- Dynamically adding and deleting criteria items
- Selecting search operators (associated to a single criterion)
- Choosing match all or match any conjunction
- Displaying in a basic or advanced mode
- Creating saved searches
- Personalizing saved searches

By default, the advanced mode of the `query` component allows the user to add and delete criteria items to the currently displayed search. However you can implement your own `QueryModel` class that can hide certain features in basic mode (and expose them only in advanced mode). For example, you might display operators only in advanced mode or display more criteria in advanced mode than in basic mode.

Typically, the results of the query are displayed in a table or tree table, which is identified using the `resultComponentId` attribute of the `query` component. However, you can display the results in any other output components as well. The component configured to display the results is automatically rerendered when a search is performed.

Figure 14-1 shows an advanced mode `query` component with three search criteria.

Figure 14-1 Query Component with Three Search Criteria

You can create *seeded searches*, that is, searches whose criteria are already determined and from which the user can choose, or you can allow the user to add criterion and then save those searches. For example, Figure 14-1 shows a seeded search for an employee. The user can enter values for the criteria on which the search will execute. The user can also choose the operands (greater than, equals, less than) and the conjunction (matches all or matches any, which creates either an "and" or "or" query). The user can click the Add Fields dropdown list to add one or more criteria and then save that search. If the application is configured to use persistence, then those search criteria, along with the chosen operands and conjunctions, can be saved and reaccessed using a given search name (for more information about persistence, see [Allowing User Customization on JSF Pages](#)).

The `quickQuery` component is a simplified version of the `query` component. The user can perform a search on any one of the searchable attributes by selecting it from a dropdown list. Figure 14-2 shows a `quickQuery` component in horizontal layout.

Figure 14-2 A QuickQuery Component in Horizontal Layout

Both the `query` and `quickQuery` components use the `QueryModel` class to define and execute searches. Create the associated `QueryModel` classes for each specific search you want users to be able to execute.

Tip:

Instead of having to build your own `QueryModel` implementation, you can use ADF Business Components, which provide the needed functionality. See *Creating ADF Databound Search Forms in [Developing Fusion Web Applications with Oracle Application Development Framework](#)*.

The `QueryModel` class manages `QueryDescriptor` objects, which define a set of search criteria. The `QueryModel` class is responsible for creating, deleting, and

updating `QueryDescriptor` objects. The `QueryModel` class also retrieves saved searches, both those that are seeded and those that the user personalizes. For information, refer to the *Java API Reference for Oracle ADF Faces*.

You must create a `QueryDescriptor` class for each set of search criteria items. The `QueryDescriptor` class is responsible for accessing the criteria and conjunction needed to build each seeded search. It is also responsible for dynamically adding, deleting, or adding and deleting criteria in response to end-user's actions. The `QueryDescriptor` class also provides various UI hints such as mode, auto-execute, and so on. For information, refer to the *Java API Reference for Oracle ADF Faces*. One `QueryModel` class can manage multiple `QueryDescriptor` objects.

When a user creates a new saved search, a new `QueryDescriptor` object is created for that saved search. The user can perform various operations on the saved search, such as deleting, selecting, resetting, and updating. When a search is executed or changed, in addition to calling the appropriate `QueryModel` method to return the correct `QueryDescriptor` object, a `QueryOperationEvent` event is broadcast during the Apply Request Values phase. This event is consumed by the `QueryOperationListener` handlers during the Invoke Application phase of the JSF lifecycle. The `QueryOperationEvent` event takes the `QueryDescriptor` object as an argument and passes it to the listener. ADF Faces provides a default implementation of the listener. For details of what the listener does, see [Table 14-2](#).

For example, updating a saved search would be accomplished by calling the `QueryModel`'s `update()` method. A `QueryOperationEvent` event is queued, and then consumed by the `QueryOperationListener` handler, which performs processing to change the model information related to the update operation.

The query operation actions that generate a `QueryOperationEvent` event are:

- Saving a search
- Personalizing a search
- Resetting a search
- Re-ordering a search
- Adding fields to a search
- Deleting a saved search
- Toggling between the basic and advanced mode
- Resetting a saved search
- Selecting a different saved search
- Updating a saved search
- Updating the value of a criterion that has dependent criteria

The `hasDependentCriterion` method of the `AttributeCriterion` class can be called to check to see whether a criterion has dependents. By default, the method returns `false`, but it returns `true` if the criterion has dependent criteria. When that criterion's value has changed, a `QueryOperationEvent` is queued for the Update Model Values JSF lifecycle phase. The model will need a listener to update the values of the dependent criterion based on the value entered in its root criteria.

Query Component Use Cases and Examples

The query component can be used in several different modes to accommodate the needs of your application. It can be configured with seeded searches and provide customization and personalization functions. The query component is a feature-rich component that can be used to implement enterprise search functions.

You can use query and quick query components to build complex transactional search forms. The query components are model-driven and provide many functional and display options. The quick query component has a small footprint and provide a simple search on one attribute. The query component has a larger footprint but provides multiple criterion searches and other search features.

Additional Functionality for the Query Components

You may find it helpful to understand other ADF Faces features before you implement your query components. Additionally, once you have added a query or quick query component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that query components can use.

- All query components have JavaScript client APIs that you can use to set or get property values. See the *JavaScript API Reference for Oracle ADF Faces*.
- You can display tips and messages, as well as associate online help with query components. See [Displaying Tips, Messages, and Help](#).
- You can change appearance of the components using skins. See [Customizing the Appearance Using Styles and Skins](#).
- You can make your query components accessible. See [Developing Accessible ADF Faces Pages](#).
- Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- If your application uses ADF Model, then you can create automatically bound search forms using data controls (whether based on ADF Business Components or other business services). See *Creating Databound Search Forms in Developing Fusion Web Applications with Oracle Application Development Framework*.

Creating the Query Data Model

An ADF query data model (QueryModel) describes a saved search. You can use a query component if you have created a query data model or by adding additional logic to a managed bean without creating a query data model.

Before you can use the query components, you must to create your QueryModel classes. For information about the query data model, see the *Java API Reference for Oracle ADF Faces*.

Tip:

You can use the `quickQuery` component without implementing a `QueryModel` class. However, you will have to add some additional logic to a managed bean. See [How to Use a quickQuery Component Without a Model](#).

Query component has a `refresh()` method on the `UIXQuery` component. This method should be called when the model definition changes and the query component need to be refreshed (i.e., all its children removed and recreated). When a new criterion is added to the `QueryDescriptor` or an existing one is removed, if the underlying model returns a different collection of criterion objects than what the component subtree expects, then this method should be called. `QueryOperationListener`, `QueryListener`, and `ActionListener` should all call this method. The query component itself will be flushed at the end of the Invoke Application Phase. This method is a no-op when called during the Render Response Phase.

To better understand what your implementations must accomplish, [Table 14-1](#) and [Table 14-2](#) map the functionality found in the UI component shown in [Figure 14-3](#) with the corresponding interface.

Figure 14-3 Query Component and Associated Popup Dialog

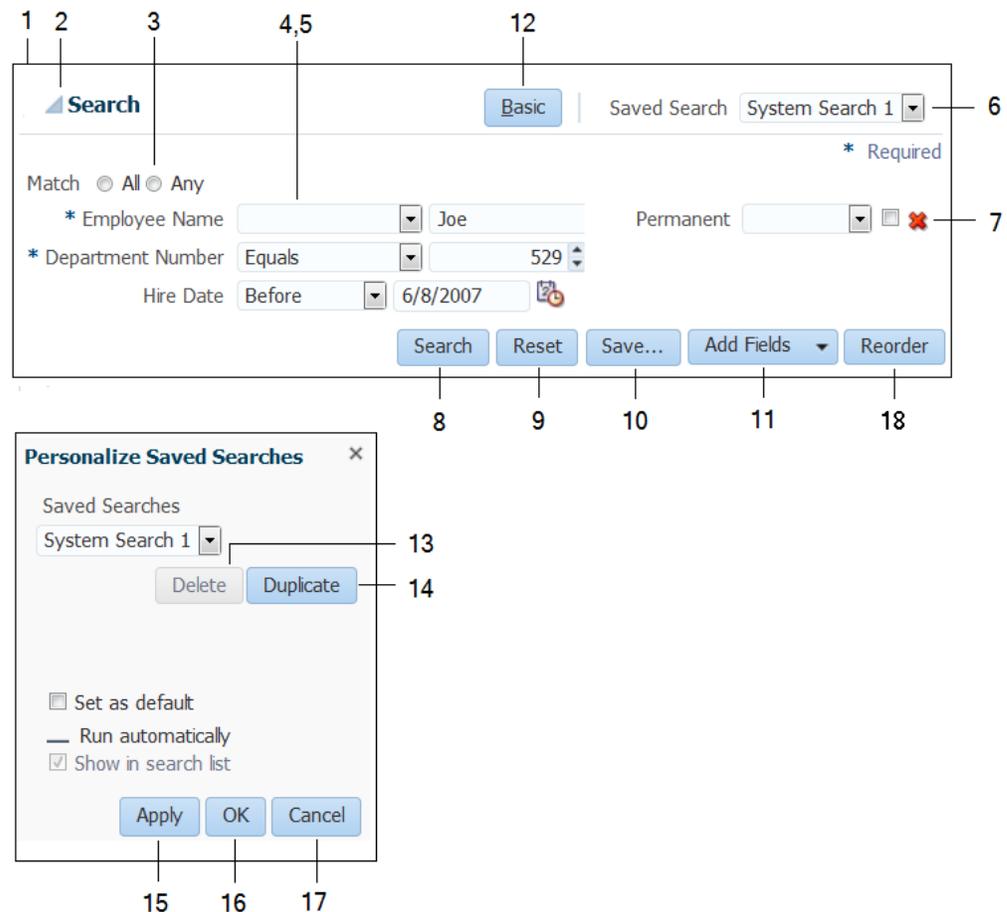


Table 14-1 shows UI artifacts rendered for the `query` component, the associated class, class property, and methods used by the artifact.

Table 14-1 Query UI Artifacts and Associated Model Class Operations and Properties

Figure Reference	UI Artifact	Class Property/Methods Used	Comments
1	Search panel	The <code>QueryDescriptor</code> instance provides the items displayed in the panel.	Based on a saved search.
2	Disclosure icon		Opens or closes the search panel
3	Match type radio button	Available through the <code>getConjunction()</code> method on the <code>ConjunctionCriterion</code> class.	<p>Displays the default conjunction to use between search fields, when a query is performed. If a default is set, and it is the same for all search fields, it appears selected. If the search fields are configured such that a mix of different conjunctions must be used between them, then a value may not be selected on the UI.</p> <p>For example, if the <code>All</code> conjunction type is used between all the search fields, then <code>All</code> appears selected. If it is a mix of <code>All</code> and <code>Any</code>, then none of the radio buttons appears selected.</p> <p>The Match Type will be read only if the <code>conjunctionReadOnly</code> property is set to <code>true</code>. Its not rendered at all when the <code>displayMode</code> attribute is set to <code>simple</code>.</p>
4	Group of search fields	The collection of search fields for a <code>QueryDescriptor</code> object is represented by a <code>ConjunctionCriterion</code> object, returned by the method <code>getConjunctionCriterion()</code> on the <code>QueryDescriptor</code> class. The <code>getCriterionList()</code> method returns a <code>List<Criterion></code> list.	Displays one or more search fields associated with the currently selected search.

Table 14-1 (Cont.) Query UI Artifacts and Associated Model Class Operations and Properties

Figure Reference	UI Artifact	Class Property/Methods Used	Comments
5	Search field	<p>An <code>AttributeCriterion</code> class provides information specific to a search field instance. An <code>AttributeCriterion</code> object is an item in the <code>List<Criterion></code> list returned by <code>getCriterionList()</code> method on the <code>ConjunctionCriterion</code> class (see #4).</p> <p>An <code>AttributeDescriptor</code> class provides static information pertaining to a search field. This is available through the method <code>getAttribute()</code>, on the <code>AttributeCriterion</code> class.</p> <p>The <code>getConverter()</code> method of the <code>AttributeDescriptor</code> class can be overridden to return a converter object of type <code>javax.faces.convert.Converter</code>. When defined, the attribute value is converted using this converter instance. The default return value is <code>null</code>.</p> <p>The <code>hasDependentCriterion</code> method in the <code>AttributeCriterion</code> class returns <code>true</code> if the criterion has dependents. If the criterion has dependents, then the dependent criterion fields are refreshed when the value for this criterion changes. By default this method returns <code>false</code>.</p>	<p>Each search field contains a label, an operator, one or more value components (for example, an input text component), and an optional delete icon. The information required to render these can be either specific to an instance of a search field (in a saved search) or it can be generic and unchanging regardless of which saved search it is part of.</p> <p>For example, assume an <code>Employee</code> business object contains the search fields <code>Employee Name</code> and <code>Salary</code>.</p> <p>A user can then configure two different searches: one named <code>Low Salaried Employees</code> and one named <code>High Salaried Employees</code>. Both searches contain two search fields based on the <code>Employee</code> and <code>Salary</code> attributes. Even though both saved searches are based on the same attributes of the <code>Employee</code> object, the search field <code>Salary</code> is configured to have its default operator as less than and value as 50000.00 for the <code>Low Salaried Employees</code> search and for the <code>High Salaried Employees</code> search, with a default operator of greater than and value of 100000.00.</p> <p>Selecting the saved searches on the UI will show the appropriate operator and values for that search.</p> <p>Regardless of the search selected by the user, the search field for <code>Salary</code> always has to render a number component, and the label always has to show <code>Salary</code>.</p>

Table 14-1 (Cont.) Query UI Artifacts and Associated Model Class Operations and Properties

Figure Reference	UI Artifact	Class Property/Methods Used	Comments
6	Saved Searches dropdown	System- and user-saved searches are available through the methods <code>getSystemQueries()</code> and <code>getUserQueries()</code> on the <code>QueryModel</code> class.	<p>Displays a list of available system- and user-saved searches. Saved searches are listed in alphabetical order and are case-insensitive.</p> <p>A Personalize option is also added if the <code>saveQueryMode</code> property is set to default. Selecting this option opens a Personalize dialog, which allows users to personalize saved searches. They can duplicate or update an existing saved search.</p> <p>When you create a new saved search based on an existing saved search and personalize the search, the <code>Set as Default</code> and <code>Run Automatically</code> settings are not selected and do not default to the settings from the original saved search. This is to ensure better performance in case if the newly created search includes all records, which should only be used in advanced search.</p>

[Table 14-2](#) shows the behaviors of the different UI artifacts, and the associated methods invoked to execute the behavior.

Table 14-2 UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
7	Delete icon	During the Invoke Application phase, the method <code>removeCriterion()</code> on the <code>QueryDescriptor</code> class is called automatically by an internal <code>ActionListener</code> handler registered with the command component.	<code>ActionEvent</code>	Deletes a search field from the current <code>QueryDescriptor</code> object.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
8	Search button	<p>During the Apply Request Values phase of the JSF lifecycle, a <code>QueryEvent</code> event is queued, to be broadcast during the Invoke Application phase.</p> <p>During the Update Model Values phase, the selected operator and the values entered in the search fields are automatically updated to the model using the EL expressions added to the operator and value components (see How to Add the Query Component). These expressions should invoke the <code>get/setOperator()</code>; <code>get/setOperators()</code>; and <code>getValues()</code> methods, respectively, on the <code>AttributeCriteria</code> class.</p> <p>During the Invoke Application phase, the <code>QueryListener</code> registered with the query component is invoked and this performs the search. You must implement this listener.</p>	<code>QueryEvent</code>	<p>Rendered always on the footer (footer contents are not rendered at all when the <code>displayMode</code> attribute is <code>simple</code>)</p> <p>Performs a query using the selected operator and the selected Match radio button (if no selection is made the default is used), and the values entered for every search field.</p>

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
9	Reset button	<p>During the Apply Request Values phase of the JSF lifecycle, a <code>QueryOperationEvent</code> event is queued with the operation type <code>QueryOperationEvent.Operation.RESET</code>, to be broadcast during the Invoke Application phase.</p> <p>During the Invoke Application phase, the method <code>reset()</code> on the <code>QueryModel</code> class is called. This is done automatically by an internal <code>QueryOperationListener</code> handler registered with the query component. You must override this method to reset the <code>QueryDescriptor</code> object to its original state.</p>	<p><code>QueryOperationEvent</code> (an internal <code>QueryOperationListener</code> handler is registered with the query component that in turn calls the methods).</p>	Resets the search fields to its previous saved state.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
10	Save button	<p>During the Apply Request Values phase of the JSF lifecycle, a <code>QueryOperationEvent</code> event is queued with the operation type <code>QueryOperationEvent.Operation.SAVE</code>, to be broadcast during the Invoke Application phase.</p> <p>During the Invoke Application phase, the method <code>create()</code> on the <code>QueryModel</code> class is called. After the call to the <code>create()</code> method, the <code>update()</code> method is called to save the hints (selected by the user in the dialog) onto the new saved search. This is done automatically by an internal <code>QueryOperationListener</code> handler registered with the query component. You must override this method to create a new object based on the argument passed in.</p>	<code>QueryOperationEvent</code> (an internal <code>QueryOperationListener</code> handler is registered with the query component that in turn calls the model methods).	Creates a new saved search based on the current saved search settings, including any new search fields added by the user.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
11	Add Fields dropdown list	During the Invoke Application phase, the method <code>addCriterion()</code> on the <code>QueryDescriptor</code> class is called automatically by an internal <code>ActionListener</code> handler registered with the command component. You must override this method to create a new <code>AttributeCriterion</code> object based on the <code>AttributeDescriptor</code> object (identified by the <code>name</code> argument).	<code>ActionEvent</code>	Adds an attribute as a search field to the existing saved search.
12	Mode (Basic or Advanced) button	During the Apply Request Values phase of the JSF lifecycle, a <code>QueryOperationEvent</code> event is queued with the operation type <code>QueryOperationEvent.Operation.MODE_CHANGE</code> , to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method <code>changeMode()</code> on the <code>QueryModel</code> class is called.	<code>QueryOperationEvent</code> (an internal <code>QueryOperationEventListener</code> handler is registered with the query component that in turn calls the model methods).	Clicking the mode button toggles the mode.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
13	Delete button	During the Invoke Application phase, the method <code>delete()</code> on the <code>QueryModel</code> class is called. This is done automatically by an internal <code>QueryOperationListener</code> handler registered with the query component. You must override this method order to delete the <code>QueryDescriptor</code> object.	<code>ActionEvent</code>	Deletes the selected saved search, unless it is the one currently in use.
14	Duplicate button	During the Apply Request Values phase of the JSF lifecycle, a <code>QueryOperationEvent</code> event is queued with the operation type <code>QueryOperationEvent.OPERATION.DUPLICATE</code> , to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method <code>update()</code> on the <code>QueryModel</code> class is called. This is done automatically by an internal <code>QueryOperationListener</code> handler registered with the query component. You must override this method in order to update the <code>QueryDescriptor</code> object using the arguments passed in.	<code>QueryOperationEvent</code> (an internal <code>QueryOperationListener</code> is registered with the query component that in turn calls the model methods).	Duplicates the selected saved search.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
15	Apply button	<p>During the Apply Request Values phase of the JSF lifecycle, a <code>QueryOperationEvent</code> event is queued with the operation type <code>QueryOperationEvent.Operation.UPDATE</code>, to be broadcast during the Invoke Application phase.</p> <p>During the Invoke Application phase, the method <code>update()</code> on the <code>QueryModel</code> class is called. This is done automatically by an internal <code>QueryOperationListener</code> handler registered with the query component. You must override this method in order to update the <code>QueryDescriptor</code> object using the arguments passed in.</p>	<p><code>QueryOperationEvent</code> (an internal <code>QueryOperationListener</code> is registered with the query component that in turn calls the model methods).</p>	Applies changes made to the selected saved search.
16	OK button	Same as the Apply button.	<p><code>QueryOperationEvent</code> (an internal <code>QueryOperationListener</code> handler is registered with the query component that in turn calls the model methods).</p>	Applies changes made to the selected saved search and the dialog is closed afterwards.

Table 14-2 (Cont.) UI Artifact Behaviors and Associated Methods

Figure Reference	UI Artifact	Class Method Invoked	Event Generated	Comments
17	Cancel button	No method defined for this action.	QueryOperationEvent (an internal QueryOperationListener handler is registered with the query component that in turn calls the model methods).	Cancels any edits made in the dialog.
18	Reorder button	No method defined for this action.	ActionEvent	Reorders the search fields in the search panel.

How to Create the Query Data Model

Begin you begin:

It may be helpful to have an understanding of the query data model. See [Creating the Query Data Model](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Query Components](#).

To create a query model classes:

1. Create implementations of each of the interface classes. Implement one `QueryModel` class and then a `QueryDescriptor` class with appropriate criteria (operators and values) for each system-seeded search. For example implementations of the different model classes for a query, see the classes located in the `oracle.adfdemo.view.query.rich` package of the ADF Faces sample application.

Note:

If your query uses composition (for example, `ConjunctionCriterion` 1..n with `AttributeCriterion/ConjunctionCriterion`), this relationship is not enforced by the abstract interfaces. Your implementation must decide whether to use composition over association, and determine how the lifecycle of these objects are managed.

2. Create a `QueryListener` handler method on a managed bean that listens for the `QueryEvent` event (this will be referenced by a button on the query component). This listener will invoke the proper APIs in the `QueryModel` to execute the query. The following example shows the listener method of a basic `QueryListener`

implementation that constructs a `String` representation of the search criteria. This `String` is then displayed as the search result.

```
public void processQuery(QueryEvent event)
{
    DemoQueryDescriptor descriptor = (DemoQueryDescriptor) event.getDescriptor();
    String sqlString = descriptor.getSavedSearchDef().toString();
    setSqlString(sqlString);
}
```

Using the quickQuery Component

The ADF Faces `quickQuery` component is a simplified version of the `query` component that allows a search on a single item. You can use a `quickQuery` component with or without a model.

The `quickQuery` component has one dropdown list that allows a user to select an attribute to search on. The available searchable attributes are drawn from your implementation of the model or from a managed bean. The user can search against the selected attribute or against all attributes.

A `quickQuery` component may be used as the starting point of a more complex search using a `query` component. For example, the user may perform a quick query search on one attribute, and if successful, may want to continue to a more complex search. The `quickQuery` component supports this by allowing you to place command components in the end facet, which you can bind to a method on a managed bean that allows the user to switch from a `quickQuery` to a `query` component.

The `quickQuery` component renders the searchable criteria in a dropdown list and then, depending on the type of the criteria chosen at runtime, the `quickQuery` component renders different criteria fields based on the attribute type. For example, if the attribute type is `Number`, it renders an `inputNumberSpinbox` component. You do not need to add these components as long as you have implemented the complete model for your query. If instead you have the logic in a managed bean and do not need a complete model, then you create the `quickQuery` component artifacts manually. See [How to Use a quickQuery Component Without a Model](#).

How to Add the quickQuery Component Using a Model

Before you begin:

It may be helpful to have an understanding of forms and subforms. See [Using the quickQuery Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Query Components](#).

You will need to complete this task:

Create a `QueryModel` class and associated classes. See [Creating the Query Data Model](#).

To add a `quickQuery` component:

1. In the Components window, from the Data Views panel, drag a **Quick Query** and drop it onto the page.

2. Expand the **Common** section of the Properties window and set the following attributes:
 - **id**: Enter a unique ID for the component.
 - **layout**: Specify if you want the component to be displayed horizontally with the criterion and value next to each other, as shown in [Figure 14-2](#), or vertically as shown in [Figure 14-4](#).

Figure 14-4 A quickQuery Component Set to Display Vertically

- **model**: Enter an EL expression that evaluates to the class that implements the `QueryModel` class, as created in [Creating the Query Data Model](#).
 - **value**: Enter an EL expression that evaluates to the class that implements the `QueryDescriptor` class, as created in [Creating the Query Data Model](#).
3. Expand the **Behavior** section and set the following attributes:
 - **conjunctionReadOnly**: Specify whether or not the user should be able to set the Match Any or Match All radio buttons. When set to `false`, the user can set the conjunction. When set to `true`, the radio buttons will not be rendered.
 - **queryListener**: Enter an EL expression that evaluates to the `QueryListener` handler you created in [Creating the Query Data Model](#).
 4. Drag and drop a table (or other component that will display the search results) onto the page. Set the results component's `PartialTriggers` with the ID of the `quickQuery` component. The value of this component should resolve to a `CollectionModel` object that contains the filtered results.
 5. If you want users to be able to click the **Advanced** link to turn the `quickQuery` component into a full `query` component, implement logic for the link component in the `End` facet of the `quickQuery` component to hide the `quickQuery` component and display the `query` component.

How to Use a quickQuery Component Without a Model

You can use the `quickQuery` component without a model, for example if all your query logic resides in a simple managed bean, including a `QueryListener` handler that will execute the search and return the results. You must manually add and bind the components required to create the complete `quickQuery` component.

Before you begin:

It may be helpful to have an understanding of forms and subforms. See [Using the quickQuery Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Query Components](#).

To add a `quickQuery` component:

1. On a managed bean, create a `valueChangeListener` handler for the `selectOneChoice` component that will display the attributes on which the user can

- search. The `valueChangeListener` handler should handle the choice for which attribute to search on.
2. On a managed bean, create the `QueryListener` handle to execute the search. This handle will use the ID of the input component used to enter the search criterion value, to retrieve the component and the value to execute the query.
 3. In the Components window, from the Data Views panel, drag a **Quick Query** and drop it onto the page.
 4. In the Properties window, expand the **Common** section, and set the following attributes:
 - **id**: Enter a unique ID for the component.
 - **layout**: Specify if you want the component to display horizontally with the criterion and value next to each other, as shown in [Figure 14-2](#), or vertically, as shown in [Figure 14-4](#).
 5. Expand the **Behavior** section and set the `QueryListener` attribute to an EL expression that evaluates to the `QueryListener` handler created in [Step 2](#).
 6. In the Components window, from the Text and Selection panel, drag a **Choice** and drop it onto the `criteriaItems` facet of the `quickQuery` component. In the dialog, choose either to enter an EL expression that evaluates to the list of attributes on which the user can search, or to enter a static list. For help with the dialog, press F1 or click **Help**.
 7. In the Structure window, select the `selectOneChoice` component in the `criteriaItems` facet, and set the following attributes:
 - **simple**: Set to `true` so that no label for the component displays.
 - **valueChangeListener**: Enter an EL expression that evaluates to the listener created in [Step 1](#).
 - **autoSubmit**: Set to `true`.
 8. From the Components window, from the Text and Selection panel, drag a **Select Item** onto the `selectOneChoice`. You can add as many as you need. For information about using the `selectOneChoice` and `selectItems` components, see [Using Selection Components](#).
 9. In the Components window, from the Text and Selection panel, drag an **inputText** component as a direct child to the `quickQuery` component. Set the following attributes:
 - **simple**: Set to `true` so that the label is not displayed.
 - **value**: Enter an EL expression that evaluates to the property that will contain the value that the user enters.

 **Tip:**

If you do not provide an `inputText` component, then at runtime, a disabled `inputText` component and a disabled Go icon will be rendered.

10. If you want users to be able to click the **Advanced** link to turn the `quickQuery` component into a full `query` component, implement logic for the link component in

the `End` facet of the `quickQuery` component to hide the `quickQuery` component and display the `query` component.

11. In the Components window, from the Data Views panel, drag a **table** (or other component that will display the search results) onto the page. Set the results component's `PartialTriggers` with the ID of the `quickQuery` component. The value of this component should resolve to a `CollectionModel` object that contains the filtered results.

What Happens at Runtime: How the Framework Renders the `quickQuery` Component and Executes the Search

When the `quickQuery` component is bound to a `QueryDescriptor` object, the `selectOneChoice` and `inputText` components are automatically added at runtime as the page is rendered. However, you can provide your own components. If you do provide both the component to display the searchable attributes and the `inputText` components, then you need the `QueryListener` handler to get the name-value pair from your components.

If you provide only your own component to show the searchable attributes (and use the default input text component), the framework will display an input text component. You must have your `QueryListener` handler get the attribute name from the dropdown list and the value from the `QueryDescriptor.getCurrentCriterion()` method to perform the query.

If you provide only your own component to collect the searchable attribute value (and use the default `selectOneChoice` component to provide the attribute name), then the framework will display the `selectOneChoice` component. You must have your `QueryListener` handler get the attribute name from the `QueryDescriptor.getCurrentCriterion()` method and the value from your component.

If you choose not to bind the `QuickQuery` component value attribute to a `QueryDescriptor` object, and you provide both components, when the **Go** button is clicked, the framework queues a `QueryEvent` event with a null `QueryDescriptor` object. The provided `QueryListener` handler then executes the query using the `changeValueListener` handler to access the name and the input component to access the value. You will need to implement a `QueryListener` handler to retrieve the attribute name from your `selectOneChoice` component and the attribute value from your `inputText` component, and then perform a query.

Using the query Component

The ADF Faces `query` components are used to perform a complete feature search or a quick query search. You can also combine the search criteria by using the `WHERE`, `AND`, `OR` clauses.

The `query` component is used for full feature searches. It has a basic and an advanced mode, which the user can toggle between by clicking a button.

The features for a basic mode query include:

- Dropdown list of selectable search criteria operators
- Selectable `WHERE` clause conjunction of either `AND` or `OR` (match all or match any)

- Saved (seeded) searches
- Personalized saved searches

The advanced mode query form also includes the ability for the user to dynamically add search criteria by selecting from a list of searchable attributes. The user can subsequently delete any criteria that were added.

The user can select from the dropdown list of operators to create a query for the search. The input fields may be configured to be list-of-values (LOV), number spinners, date choosers, or other input components.

Note:

If you want a uniform width for all the operator dropdown fields in the search panel, you can set the following skinning property:

```
af|query {
  -tr-operator-size: constant;
}
```

Having a uniform operator field width would vertically align the operator and search boxes for all the search fields, which results in a more organized appearance.

For a general overview of skins, see [Customizing the Appearance Using Styles and Skins](#).

To support selecting multiple items from a list, the model must expose a control hint on `viewCriteriaItem` and the underlying attribute must be defined as an LOV in the corresponding view object. The hint is used to enable or disable the multiple selection or "in" operator functionality. When multiple selection is enabled, selecting the `Equals` or `Does not equal` operator will render the search criteria field as a `selectManyChoice` component. The user can choose multiple items from the list.

The component for the search criteria field depends on the underlying attribute data type, the operator that was chosen, and whether multiple selection is enabled. For example, a search field for an attribute of type `String` with the `Contains` operator chosen would be rendered as an `inputText` component, as shown in [Table 14-3](#).

If the operator is `Equals` or `Does not equal`, but multiple selection is not enabled, the component defaults to the component specified in the Default List Type hint from the model.

Table 14-3 Rendered Component for Search Criteria Field of Type String

Operator	Component	Component When Multiple Select Is Enabled
Starts with	<code>af:inputText</code>	<code>af:inputText</code>
Ends with	<code>af:inputText</code>	<code>af:inputText</code>
Equals	Default list type hint	<code>af:selectManyChoice</code>
Does not equal	Default list type hint	<code>af:selectManyChoice</code>

Table 14-3 (Cont.) Rendered Component for Search Criteria Field of Type String

Operator	Component	Component When Multiple Select Is Enabled
Less than	af:inputText	af:inputText
Greater than	af:inputText	af:inputText
Less than or equal to	af:inputText	af:inputText
Greater than or equal to	af:inputText	af:inputText
Between	af:inputText	af:inputText
Not Between	af:inputText	af:inputText
Contains	af:inputText	af:inputText
Does not contain	af:inputText	af:inputText
Contains	af:inputText	af:inputText
Does not contain	af:inputText	af:inputText
Is blank	None	None
Is not blank	None	None

If the underlying attribute is the `Number` data type, the component that will be rendered is shown in [Table 14-4](#).

Table 14-4 Rendered Component for Search Criteria Field of Type Number

Operator	Component	Component When Multiple Select Is Enabled
Equals	Default list type hint	af:selectManyChoice
Does not equal	Default list type hint	af:selectManyChoice
Less than	af:inputNumberSpinBox	af:inputNumberSpinBox
Less than or equal to	af:inputNumberSpinBox	af:inputNumberSpinBox
Greater than	af:inputNumberSpinBox	af:inputNumberSpinBox
Greater than or equal to	af:inputNumberSpinBox	af:inputNumberSpinBox
Between	af:inputNumberSpinBox	af:inputNumberSpinBox
Not between	af:inputNumberSpinBox	af:inputNumberSpinBox
Is blank	None	None
Is not blank	None	None

If the underlying attribute is the `Date` data type, the component that will be rendered is shown in [Table 14-5](#).

Table 14-5 Rendered Component for Search Criteria Field of Type Date

Operator	Component	Component When Multiple Select Is Enabled
Equals	Default list type hint	af:selectManyChoice
Does not equal	Default list type hint	af:selectManyChoice
Before	af:inputDate	af:inputDate
After	af:inputDate	af:inputDate
On or before	af:inputDate	af:inputDate
On or after	af:inputDate	af:inputDate
Between	af:inputDate (2)	af:inputDate (2)
Not between	af:inputDate (2)	af:inputDate (2)
Is blank	None	None
Is not blank	None	None

If a search criterion's underlying attribute was defined as an LOV, in order for the auto-complete feature to work, the `ListOfValues` model instance returned by the `getModelList` method of the `AttributeCriterion` class must return `true` for its `isAutoCompleteEnabled` method. For information about LOV, see [Using List-of-Values Components](#).

When `autoSubmit` is set to `true`, any value change on the search criterion will be immediately pushed to the model. The query component will automatically flush its criterion list only when it has dependent criteria. If the criterion instance has no dependent criteria but `autoSubmit` is set to `true`, then the query component will be only partially refreshed.

A **Match All** or **Match Any** radio button group further modifies the query. A **Match All** selection is essentially an `AND` function. The query will return only rows that match all the selected criteria. A **Match Any** selection is an `OR` function. The query will return all rows that match any one of the criteria items.

After the user enters all the search criteria values (including null values) and selects the **Match All** or **Match Any** radio button, the user can click the **Search** button to initiate the query. The query results can be displayed in any output component. Typically, the output component will be a table or tree table, but you can associate other display components such as `af:forms`, `af:outputText`, and graphics to be the results component by specifying it in the `resultComponentId` attribute.

If the **Basic** or **Advanced** button is enabled and displayed, the user can toggle between the two modes. Each mode will display only the search criteria that were defined for that mode. A search criteria field can be defined to appear only for basic, only for advanced, or for both modes.

In advanced mode, the control panel also includes an **Add Fields** button that exposes a popup list of searchable attributes. When the user selects any of these attributes, a dynamically generated search criteria input field and dropdown operator list is displayed. The position of all search criteria input fields, as well as newly added fields, are determined by the model implementation.

This newly created search criteria field will also have a delete icon next to it. The user can subsequently click this icon to delete the added field. The originally defined search criteria fields do not have a delete icon and therefore cannot be deleted by the user. [Figure 14-5](#) shows an advanced mode `query` component with a dynamically added search criteria field named Salary. Notice the delete icon (an X) next to the field.

Figure 14-5 Advanced Mode Query with Dynamically Added Search Criteria

The screenshot displays the 'Advanced Mode Query' interface. At the top, there is a 'Search' header with a 'Basic' mode button and a 'Saved Search' dropdown menu currently showing 'System Search 1'. A note indicates that fields marked with an asterisk are required. Below the header, there are radio buttons for 'Match' set to 'All'. The 'General' section contains two required fields: '* Employee Name' with the value 'Joe' and '* Department Number' with the operator 'Equals' and the value '529'. The 'Confidential' section includes 'Hire Date' set to 'Before' with the value '6/8/2007' and a calendar icon, and 'Show Salary' with a delete icon (an X). At the bottom, there are five buttons: 'Search', 'Reset', 'Save...', 'Add Fields', and 'Reorder'.

The user can also save the entered search criteria and the mode by clicking the **Save** button. A popup dialog allows the user to provide a name for the saved search and specify hints by selecting checkboxes. A persistent data store is required if the saved search is to be available beyond the session. For information about persistence, see [Allowing User Customization on JSF Pages](#).

A seeded search is essentially a saved search that was created by the application developer. When the component is initialized, any seeded searches associated with that `query` component become available for the user to select.

Any user-created saved searches and seeded system searches appear in the Saved Search dropdown list. The seeded searches and user-saved searches are separated by a divider. They are sorted alphabetically and are case insensitive.

Users can also personalize the saved and seeded searches for future use. Personalization of saved searches requires the availability of a persistent data store. For information about persistence, see [Allowing User Customization on JSF Pages](#).

Along with the default display described previously, you can also configure the `query` component to display in a compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown list moves next to the expand or collapse icon. [Figure 14-6](#) shows the same `query` component as in [Figure 14-5](#), but set to compact mode.

Figure 14-6 Query Component in Compact Mode

The simple mode displays the component without the header and footer, and without the buttons typically displayed in those areas. [Figure 14-7](#) shows the same query component set to simple mode.

Figure 14-7 Query Component in Simple Mode

The query component supports `toolbar` and `footer` facets that allow you to add additional components to the query, such as buttons. For example, you can create command components to toggle between the `quickQuery` and `query` components and place those in a toolbar in the `toolbar` facet.

Because the query component is responsible for rendering its subcomponents (input fields, selection list, buttons, etc.), you should not use `inlineStyle` with the query. If you use `inlineStyle`, it may result in unexpected display behavior.

How to Add the Query Component

Before you begin:

It may be helpful to have an understanding of forms and subforms. See [Using the query Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Query Components](#).

You will need to complete this task:

- Create a `QueryModel` class and associated classes. See [Creating the Query Data Model](#).

To add a query component:

1. In the Components window, from the Data Views panel, drag a **Query** and drop it onto the page.
 2. In the Properties window, expand the **Common** section and set the following attributes:
 - **id**: Set a unique ID for the component.
 - **model**: Enter an EL expression that resolves to the `QueryModel` class, as created in [Creating the Query Data Model](#).
 - **value**: Enter an EL expression that resolves to the `QueryDescriptor` class, as created in [Creating the Query Data Model](#).
 3. Expand the **Appearance** section and set the following attributes:
 - **displayMode**: Specify if you want the component to display in Default, Simple, or Compact mode.
 - **saveQueryMode**: Specify if you want saved searches to be displayed and used at runtime. Set to `default` if you want the user to be able to view and edit all saved searches. Set to `readOnly` if you want the user to only be able to view and select saved searches, but not update them. Set to `hidden` if you do not want any saved searches to be displayed.
-  **Note:**
- You can set the `oracle.adf.faces.component.query.saveQueryMode` context parameter in `web.xml` file to control the `saveQueryMode` setting globally, at the application level. At runtime, ADF applies the setting in the `web.xml` file to all `af:query` components, where the `saveQueryMode` attribute setting on a specific query component overrides the global setting. See [Save Query Mode](#).
- **modeButtonPosition**: Specify if you want the button that allows the user to switch the mode from basic to advanced to be displayed in toolbar (the default) or in the `footer` facet.
 - **modeChangeVisible**: Set to `false` if you want to hide the basic or advanced toggle button.
4. Expand the **Behavior** section and set the following:
 - **conjunctionReadOnly**: Set to `false` if you want the user to be able to select a radio button to determine if the search should match all criteria (query will use the `AND` function) or any criteria (query will use the `OR` function). When set to `true`, the radio buttons will not be rendered.
 - **queryListener**: Enter an EL expression that evaluates to the `QueryListener` handler, as created in [Creating the Query Data Model](#).
 5. Expand the **Other** section and set the following:
 - **CriterionFeatures**: Set to `matchCaseDisplayed` will require all string-based search criterion to be case-sensitive. Set to `requiredDisplayed` will require all criterion be displayed.

- **runQueryAutomatically:** Select `allSavedSearches` to enable all system and user-created saved searches to run automatically upon initial render, changes in saved search selection, and reset.

Select `searchDependent` to allow the developer to choose the **Run Automatically** option at design time for each system query. Default is `searchDependent`.

For new user-created saved searches, if `searchDependent` is selected, the Create Saved Search dialog will have the **Run Automatically** option selected by default. If `allSavedSearches` is selected, the **Run Automatically** option is not displayed but is set to true implicitly.

6. In the Components window, from the Data Views panel, drag a **table** (or other component that will display the search results) onto the page. Set an ID on the table. The value of this component should resolve to a `CollectionModel` object that contains the filtered results.
7. In the Structure window, select the `query` component and set the `resultComponentID` to the ID of the table.

15

Using Menus, Toolbars, and Toolboxes

This chapter describes how to create menus and toolbars using the ADF Faces `menu`, `menuBar`, `commandMenuItem`, `goMenuItem`, `toolbar`, and `toolbox` components. For information about creating navigation menus, that is, menus that allow you to navigate through a hierarchy of pages, see [Using Navigation Items for a Page Hierarchy](#).

This chapter includes the following sections:

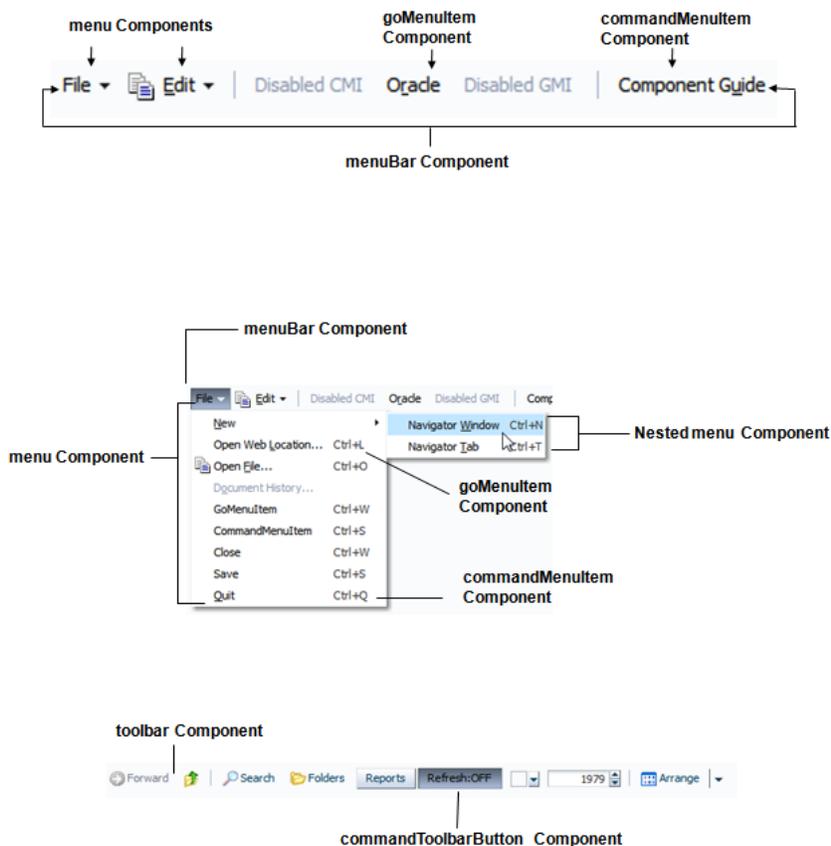
- [About Menus, Toolbars, and Toolboxes](#)
- [Using Menus in a Menu Bar](#)
- [Using Toolbars](#)

About Menus, Toolbars, and Toolboxes

ADF Faces provides menus and toolbars as UI elements that allow a user to choose from a specified list of options (in the case of a menu) or to click buttons (in the case of a toolbar) to effect some change to the application. You can select any menu dropdown or buttons in a toolbar to trigger an action.

Menu bars and toolbars allow you to organize menus, buttons, and other simple components in a horizontal bar. When a user clicks a menu in the bar, the menu drops down and the user can select from the menu items, which then causes some action to happen in the application. Icons in the toolbar also cause some action to happen in the application. [Figure 15-1](#) shows the different components used to create menus and toolbars.

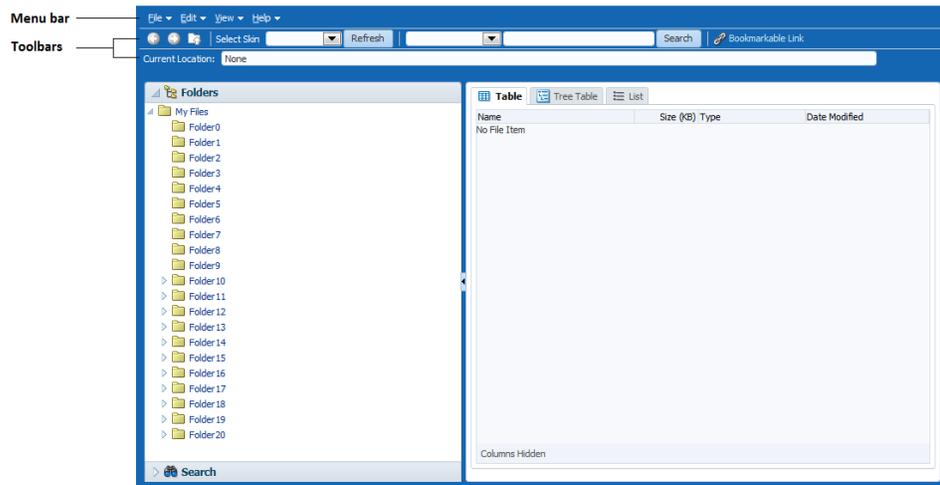
Figure 15-1 Menu and Toolbar Components



Menu Components Use Cases and Examples

Menu components are used to create menus that allow users to add or edit items, search data, change the view, or launch help. For example, the ADF Faces Components Demo application contains both a menu bar and a toolbar, as shown in [Figure 15-2](#).

Figure 15-2 Menu Bar and Toolbar in File Explorer Application



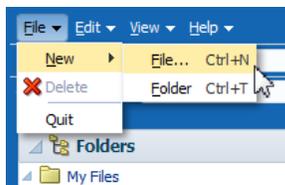
When a user chooses a menu item in the menu bar, the menu component displays a list of menu items, as shown in [Figure 15-3](#).

Figure 15-3 Menu in the File Explorer Application



As shown in [Figure 15-4](#), menus can be nested.

Figure 15-4 Nested Menu Items



Buttons in a toolbar also allow a user to invoke some sort of action on an application or to open a context menu that behaves the same as a standard menu.

You can organize toolbars and menu bars using a toolbox. The toolbox gives you the ability to define relative sizes for the toolbars on the same line and to define several layers of toolbars and menu bars vertically.

 **Note:**

If you want to create menus and toolbars in a table, then follow the procedures in [Displaying Table Menus, Toolbars, and Status Bars](#).

If you want to create a context menu for a component (that is, a menu that launches when a user right-clicks the component), follow the procedures in [How to Create a Context Menu](#).

Additional Functionality for Menu and Toolbar Components

You may find it helpful to understand other ADF Faces features before you implement your menu and toolbar components. Additionally, once you have added these components to the page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that menu and toolbar components can use.

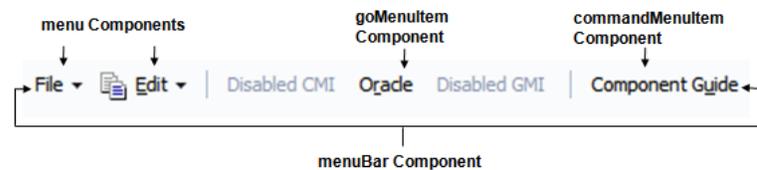
- **Invoking functionality:** ADF Faces offer tags that can be used with menu command components to invoke functionality, such as downloading a file or resetting submitted values. See [Using Buttons or Links to Invoke Functionality](#).
- **Table menus:** You can create menus and toolbars that display above a table and work only on that table (as opposed to the whole application). See [Displaying Table Menus, Toolbars, and Status Bars](#).
- **Context menus:** You can create menus that launch in a popup when a user right-clicks an item in the UI. See [How to Create a Context Menu](#).
- **Using parameters in text:** You can use the ADF Faces EL format tags if you want text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Events:** You can use command menu components to launch action events. For information about events, see [Handling Events](#). For information about action events specifically, see [Using Buttons and Links for Navigation](#).
- **Accessibility:** You can use specific attributes on the menu components to create shortcuts that allow users to open menus using a keyboard. For information about how to define access keys, see [How to Define Access Keys for an ADF Faces Component](#). For information about accessibility, see [Specifying Component-Level Accessibility Properties](#).
- **Localization:** Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- **Skins:** You can change the look and feel of menus (such as the icon used to display a selected menu item), along with some basic functionality (such as the maximum number of menu items that display) by changing the skin. See [Customizing the Appearance Using Styles and Skins](#).

Using Menus in a Menu Bar

ADF Faces provides a `menuBar` component that contains various menu items from which a user can execute a command. You can group multiple menu bars in a toolbox and can also nest `menu` components to create submenus.

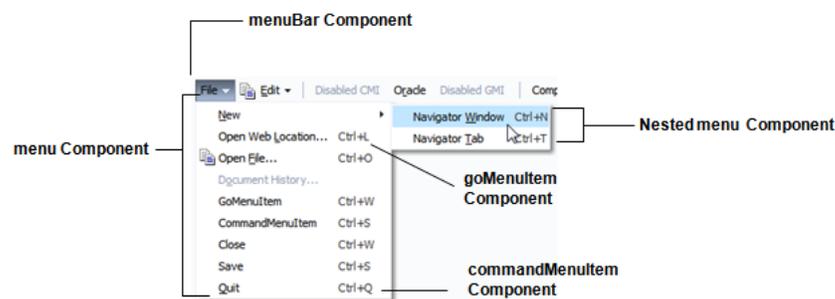
Use the `menuBar` component to render a bar that contains the menu bar items (such as **File** in the File Explorer application). These items can be `menu` components, which hold a vertical menu, as well as `commandMenuItem` components that invoke some operation on the application, and `goMenuItem` components that invoke a URL, as shown in [Figure 15-5](#).

Figure 15-5 menuBar and Child Components



Menu components can also contain `commandMenuItem` or `goMenuItem` components, or you can nest `menu` components inside `menu` components to create submenus. The different components used to create a menu are shown in [Figure 15-6](#).

Figure 15-6 Components Used to Create a Menu



You can use more than one menu bar by enclosing them in a toolbox. Enclosing them in a toolbox stacks the menu bars so that the first menu bar in the toolbox is displayed at the top, and the last menu bar is displayed at the bottom. When you use more than one menu bar in a single toolbox row (by having them grouped inside the toolbox), then the `flex` attribute will determine which menu bar will take up the most space.

If you want menu bars to be displayed next to each other (rather than being stacked), you can enclose them in a `group` component.

 **Tip:**

You can also use the `toolbox` component to group menu bars with toolbars, or to group multiple menu bars. Use the `group` component to group menu bars and toolbars on the same row.

Within a menu bar, you can set one component to stretch so that the menu bar will always be the same size as its parent container. For example, in [Figure 15-7](#), the menu bar is set to stretch a spacer component that is placed between the Disabled GMI menu and the Component Guide button. When the window is resized, that spacer component either stretches or shrinks so that the menu bar will always be the same width as the parent. Using a spacer component like this also ensures that any components to the right of the spacer will remain right-justified in the menu bar.

Figure 15-7 Spacer Component Stretches and Shrinks



When a window is resized such that all the components within the menu bar can no longer be displayed, the menu bar displays an overflow icon, identified by the arrow cursor as shown in [Figure 15-8](#).

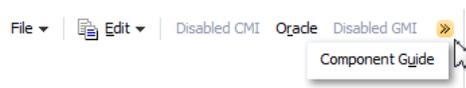
Figure 15-8 Overflow Icon in a Menu Bar

 **Note:**

The overflow icon does not display in the Internet Explorer browser. In Firefox and Chrome, the overflow icon displays only after the user resizes the browser window.

Clicking that overflow icon displays the remaining components in a popup window, as shown in [Figure 15-9](#).

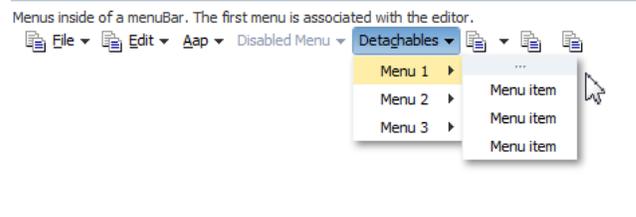
Figure 15-9 menu Component in an Overflow Popup Window



Menus and submenus can be made to be detachable and to float on the browser window. [Figure 15-10](#) shows the **Menu 1** submenu in the **Detachables** menu

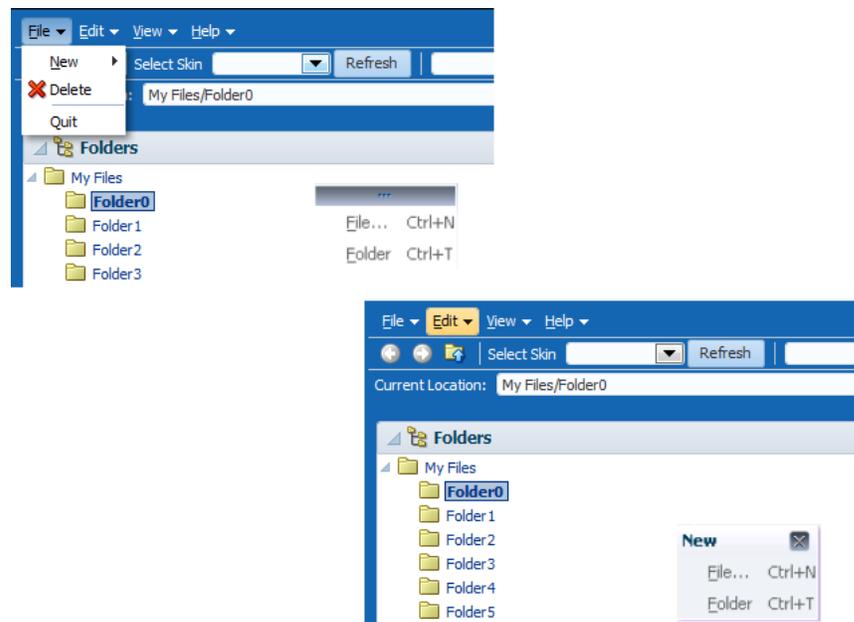
configured to be detachable. The top of the menu is rendered with a bar to indicate that it can be detached.

Figure 15-10 Detachable Menu



The user can drag the detachable menu to anywhere within the browser. When the mouse button is released, the menu stays on top of the application until the user closes it, as shown in [Figure 15-11](#).

Figure 15-11 Floating Detached Menu



 **Tip:**

Consider using detachable menus when you expect users to:

- Execute similar commands repeatedly on a page.
- Execute similar commands on different rows of data in a large table, tree table, or tree.
- View data in long and wide tables, tree tables, or trees. Users can choose which columns or branches to hide or display with a single click.
- Format data in long or wide tables, tree tables, or trees.

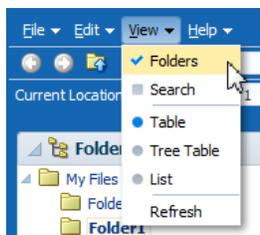
The `menu` and `commandMenuItem` components can each include an icon image. [Figure 15-12](#) shows the **Delete** menu item configured to display a delete icon (a red X).

Figure 15-12 Icons Can Be Used in Menus



Aside from always displaying graphics, you can configure `commandMenuItem` components to display a graphic when the menu item is chosen. For example, you can configure a `commandMenuItem` component to display a checkmark when chosen, or you can group menu items together and configure them to behave like a group of radio buttons, so that an icon displays next to the label when one of items in the group is chosen. [Figure 15-13](#) shows the **View** menu with the **Folders** menu item configured to use a checkmark when chosen. The **Table**, **Tree Table**, and **List** menu items are configured to be radio buttons, and allow the user to choose only one of the group.

Figure 15-13 Icons and Radio Buttons Denote the Chosen Menu Items



You can set the initial selected state of the `commandMenuItem` component using an EL expression. However, after that, the value of the selected state must be managed

through the `commandMenuItem` component using an action listener. Thus, runtime selection of the menu item will have no effect on the EL-specified property used to set the initial state.

You can also configure a `commandMenuItem` component to have an antonym. Antonyms display different text when the user chooses a menu item.

Figure 15-14 The Edit Menu of the File Explorer Application

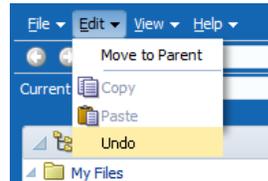


Figure 15-14 shows the `commandMenuItem` component for the **Undo** menu item configured to be an antonym. When the user chooses **Undo**, the next time the user returns to the menu, the menu item will display the antonym **Restore**, as shown in Figure 15-15.

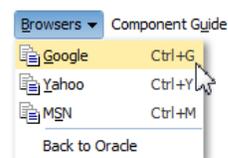
Figure 15-15 Menu Items Can Be Antonyms



Because an action is expected when a user chooses a menu item, you must bind the `action` or `actionListener` attribute of the `commandMenuItem` component to some method that will execute the needed functionality.

Along with `commandMenuItem` components, a menu can also include one or more `goMenuItem` components. These are navigation components similar to the `link` component, in that they perform direct page navigation, without delivering an `ActionEvent` event. Figure 15-16 shows three `goMenuItem` components used to navigate to external web sites.

Figure 15-16 Menus Can Use `goMenuItem` Components



Aside from menus that are invoked from menu bars, you can also create context menus that are invoked when a user right-clicks a UI component, and popup menus

that are invoked when a user clicks a command component. See [How to Create a Context Menu](#).

 **Note:**

ADF Faces provides a button with built-in functionality that allows a user to view a printable version of the current page. Menus and menu bars do not render on these pages. See [Using ADF Faces Client Behavior Tags](#).

By default, the contents of the menu are delivered immediately, as the page is rendered. If you plan on having a large number of children in a menu (multiple `menu` and `commandMenuItem` components), you can choose to configure the menu to use **lazy content delivery**. This means that the child components are not retrieved from the server until the menu is accessed.

 **Note:**

Content delivery for menus used as popup context menus is determined by the parent popup dialog, and not the menu itself.

You can also create menus that mainly provide navigation throughout the application, and are not used to cause any change on a selected item in an application. To create this type of menu, see [Using a Menu Model to Create a Page Hierarchy](#).

How to Create and Use Menus in a Menu Bar

To create a menu, you first have to create a menu bar to hold the menus. You then add and configure `menu` and `commandMenuItem` components as needed.

 **Note:**

If you want to create menus in a table, follow the procedures outlined in [Displaying Table Menus, Toolbars, and Status Bars](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Menus in a Menu Bar](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Menu and Toolbar Components](#).

To create and use menus in a menu bar:

1. If you plan on using more than one menu bar or a combination of toolbars and menu bars, create a `toolbox` component by dragging and dropping a **Toolbox** from the Menus and Toolbars panel of the Components window.

**Tip:**

The `panelHeader`, `showDetailHeader`, and `showDetailItem` components support a `toolbar` facet for adding toolboxes and toolbars to section headers and accordion panel headers.

2. Create a menu bar by dragging and dropping a **Menu Bar** from the Components window. If you are using a `toolbox` component, the **Menu Bar** should be dropped as a direct child of the `toolbox` component.

**Tip:**

Toolboxes also allow you to use the `iterator`, `switcher`, and `group` components as direct children, providing these components wrap child components that would usually be direct children of the `toolbox`. For information about toolboxes, see [Using Toolbars](#).

3. If grouping more than one menu bar within a toolbox, for each menu bar, expand the **Appearance** section and set the `flex` attribute to determine the relative sizes of each of the menu bars. The higher the number given for the `flex` attribute, the longer the toolbox will be. For the set of menu bars shown in the following example, `menuBar2` will be the longest, `menuBar4` will be the next longest, and because their `flex` attributes are not set, the remaining menu bars will be the same size and shorter than `menuBar4`.

```
<af:toolbox>
  <af:menuBar id="menuBar1" flex="0">
    <af:menu text="MenuA"/>
  </af:menuBar>
  <af:menuBar id="menuBar2" flex="2">
    <af:menu text="MenuB"/>
  </af:menuBar>
  <af:menuBar id="menuBar3" flex="0">
    <af:menu text="MenuC"/>
  </af:menuBar>
  <af:menuBar id="menuBar4" flex="1">
    <af:menu text="MenuD"/>
  </af:menuBar>
</af:toolbox>
```

**Performance Tip:**

At runtime, when available browser space is less than the space needed to display the contents of the toolbox, ADF Faces automatically displays overflow icons that enable users to select and navigate to those items that are out of view. The number of child components within a `toolbox` component, and the complexity of the children, will affect the performance of the overflow. You should set the size of the `toolbox` component to avoid overflow when possible. See [What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined](#).

 **Tip:**

You can use the `group` component to group menu bars (or menu bars and toolbars) that you want to appear on the same row. If you do not use the `group` component, the menu bars will appear on subsequent rows.

For information about how the `flex` attribute works, see [What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined](#).

4. Insert the desired number of `menu` components into the menu bar by dragging a **Menu** from the Components window, and dropping it as a child to the `menuBar` component.

You can also insert `commandMenuItem` components directly into a menu bar by dragging and dropping a **Menu Item** from the Menus and Toolbars panel of the Components window. Doing so creates a `commandMenuItem` component that renders similar to a toolbar button.

 **Tip:**

Menu bars also allow you to use the `iterator`, `switcher`, and `group` components as direct children, providing these components wrap child components that would usually be direct children of the menu bar.

5. For each `menu` component, expand the Appearance section in the Properties window and set the following attributes:
 - **Text:** Enter text for the menu's label. If you want to also provide an access key (a letter a user can use to access the menu using the keyboard), then leave this attribute blank and enter a value for `textAndAccessKey` instead.
 - **TextAndAccessKey:** Enter the menu label and access key, using conventional ampersand notation. For example, `&File` sets the menu label to **File**, and at the same time sets the menu access key to the letter **F**. For information about access keys and the ampersand notation, see [Specifying Component-Level Accessibility Properties](#).
 - **Icon:** Use the dropdown list to select the icon. If the icon does not display in this menu, use the dropdown menu to the right of the list to choose **Edit**, and browse to select the icon.
6. If you want the menu to be detachable (as shown in [Figure 15-10](#)), expand the Behavior section in the Properties window and set the **Detachable** attribute to `true`. At runtime, the user can drag the menu to detach it, and drop it anywhere on the screen (as shown in [Figure 15-11](#)).
7. If you want the menu to use lazy content delivery, set the **ContentDelivery** attribute to `lazy`.

 **Note:**

If you use lazy content delivery, any accelerators set on the child `commandMenuItem` components will not work because the contents of the menu are not known until the menu is accessed. If your menu must support accelerators, then **ContentDelivery** must be set to `immediate`.

 **Note:**

If the menu will be used inside a popup dialog or window, leave **ContentDelivery** set to `immediate`, because the popup dialog or window will determine the content delivery for the menu.

8. To create a menu item that invokes some sort of action along with navigation, drag a **Menu Item** from the Components window and drop it as a child to the `menu` component to create a `commandMenuItem` component. Create a number of `commandMenuItem` components to define the items in the vertical menu. If necessary, you can wrap the `commandMenuItem` components within a `group` component to display the items as a group.

The following example shows simplified code for grouping the **Folders** and **Search** menu items in one group, the **Table**, **Tree Table** and **List** menu items in a second group, and the **Refresh** menu item by itself at the end.

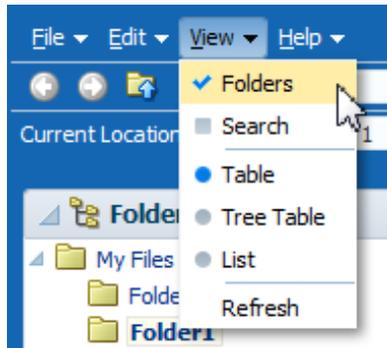
```
<af:menu id="viewMenu"
  <af:group>
    <af:commandMenuItem type="check" text="Folders"/>
    <af:commandMenuItem type="check" text="Search"/>
  </af:group>
  <af:group>
    <af:commandMenuItem type="radio" text="Table"/>
    <af:commandMenuItem type="radio" text="Tree Table"/>
    <af:commandMenuItem type="radio" text="List"/>
  </af:group>
  <af:commandMenuItem text="Refresh"/>
</menu>
```

Figure 15-17 shows how the menu will be displayed when it is first accessed.

 **Note:**

You can change the skin that ADF Faces components use. See [Customizing the Appearance Using Styles and Skins](#).

Figure 15-17 Grouped Menu Items Using the Type Attribute



 **Tip:**

By default, only up to 14 items are displayed in the menu. If more than 14 items are added to a menu, the first 14 are displayed along with a scrollbar, which can be used to access the remaining items. If you wish to change the number of visible items, edit the `af|menu {-tr-visible-items}` skinning key. See [Customizing the Appearance Using Styles and Skins](#).

You can also insert another `menu` component into an existing `menu` component to create a submenu (as shown in [Figure 15-4](#)).

 **Tip:**

Menus also allow you to use the iterator and switcher components as direct children, providing these components wrap child components that would usually be direct children of the menu.

9. For each `commandMenuItem` component, expand the Common section in the Properties window and set the following attributes:
 - **Type:** Specify a type for this menu item. When a menu item type is specified, ADF Faces adds a visual indicator (such as a radio button) and a toggle behavior to the menu item. At runtime, when the user selects a menu item with a specified type (other than the default), ADF Faces toggles the visual indicator or menu item label. Use one of the following acceptable `type` values:
 - **default:** Assigns no type to this menu item. The menu item is displayed in the same manner whether or not it is chosen.
 - **antonym:** Toggles the menu item label. The value set in the **SelectedText** attribute is displayed when the menu item is chosen, instead of the menu item defined by the value of `text` or `textAndAccessKey` attribute (which is what is displayed when the menu item is not chosen). If you select this type, you must set a value for **SelectedText**.

- **check:** Depending on the skin, toggles a square or check mark next to the menu item label. The square is displayed as solid blue when the menu item is chosen, and greyed out when not.
- **radio:** Toggles a radio button next to the menu item label. The radio button is displayed as a solid blue circle when the menu item is chosen, and greyed out when not.
- **Text:** Enter text for the menu item's label. If you wish to also provide an access key (a letter a user can use to access the item using the keyboard), then leave this attribute blank and enter a value for **TextAndAccessKey** instead. Or, you can set the access key separately using the `accessKey` attribute.
- **Selected:** Set to `true` to have this menu item appear to be chosen. The `selected` attribute is supported for check-, radio-, and antonym-type menu items only.

 **Note:**

You may use an EL expression that evaluates to an action method in a managed bean to determine the initial selection state of the `commandMenuItem`. However, at runtime, the selected state of the menu item is managed through the component and therefore requires an action listener to update the selected state property of the managed bean. For example, your managed bean might define the action listener as:

```
public void actLsn(ActionEvent ae) {  
    RichCommandMenuItem menuItem = (RichCommandMenuItem)  
    ae.getComponent();  
    val1 = menuItem.isSelected();  
}
```

- **SelectedText:** Set the alternate label to display for this menu item when the menu item is chosen. This value is ignored for all types except `antonym`.
10. Expand the Appearance section and set the following attributes:
- **Icon:** Use the dropdown list to select the icon. If the icon does not display in this menu, use the dropdown menu to the right of the list to choose **Edit**, and browse to select the icon.
 - **Accelerator:** Enter the keystroke that will activate this menu item's command when the item is chosen, for example, `Control N`. ADF Faces converts the keystroke and displays a text version of the keystroke (for example, `Ctrl+N`) next to the menu item label, as shown in [Figure 15-4](#).

 **Note:**

If you choose to use lazy content delivery, any accelerators set on the child `commandMenuItem` components will not work because the contents of the menu are not known until it is accessed. If your menu must support accelerator keys, then the `contentDelivery` attribute must be set to `immediate`.

- **TextAndAccessKey:** Enter the menu item label and access key, using conventional ampersand notation. For example, `&#x26;Save` sets the menu item label to **Save**, and at the same time sets the menu item access key to the letter `s`. For information about access keys and the ampersand notation, see [Specifying Component-Level Accessibility Properties](#).
11. Expand the Behavior section and set the following attributes:
- **Action:** Use an EL expression that evaluates to an action method in an object (such as a managed bean) that will be invoked when this menu item is chosen. The expression must evaluate to a public method that takes no parameters, and returns a `java.lang.Object` object.

If you want to cause navigation in response to the action generated by `commandMenuItem` component, instead of entering an EL expression, enter a static action outcome value as the value for the `action` attribute. You then must either set the `partialSubmit` attribute to `false`, or use a `redirect`. For information about configuring navigation in your application, see [Defining Page Flows](#).
 - **ActionListener:** Specify the expression that refers to an action listener method that will be notified when this menu item is chosen. This method can be used instead of a method bound to the `action` attribute, allowing the `action` attribute to handle navigation only. The expression must evaluate to a public method that takes an `ActionEvent` parameter, with a return type of `void`.
12. To create a menu item that simply navigates (usually to an external site), drag and drop a **Menu Item (Go)** from the Components window as a child to the menu.
13. In the Properties window, expand the **Common** section and set the following attributes:
- **Text:** Enter the text for the link.

 **Tip:**

Instead, you can use the `textAndAccessKey` attribute to provide a single value that defines the label and the access key to use for the link. For information about how to define access keys, see [How to Define Access Keys for an ADF Faces Component](#).

- **Destination:** Enter the URI of the page to which the link should navigate. For example, to navigate to the Oracle Corporation Home Page, you would enter `http://www.oracle.com`.

- **TargetFrame:** Use the dropdown list to specify where the new page should display. Choose one of the following values:
 - `_blank`: The link opens the document in a new window.
 - `_parent`: The link opens the document in the window of the parent. For example, if the link appeared in a dialog, the resulting page would render in the parent window.
 - `_self`: The link opens the document in the same page or region.
 - `_top`: The link opens the document in a full window, replacing the entire page.
14. If you want a menu bar to stretch so that it equals the width of the containing parent component, select the `menuBar` component in the Structure window, then expand the **Appearance** section in the Properties window and set **StretchId** to be the ID of the component within the menu bar that should be stretched so that the menu bar is the same size as the parent. This one component will stretch, while the rest of the components in the menu bar remain a static size.

You can also use the `stretchId` attribute to justify components to the left and right by inserting a `spacer` component, and setting that component ID as the `stretchId` for the menu bar, as shown in the following example.

```
<af:menuBar binding="{editor.component}" id="menuBar1" stretchId="stretch1">
  <af:menu text="File" id="m1">
    . . .
  </af:menu>
  . . .
  <af:commandMenuItem text="Disabled CMI"/>
  <af:goMenuItem textAndAccessKey="O&acracle destination="http://
www.oracle.com"
                    id="gm1"/>
  <af:goMenuItem text="Disabled GMI" destination="http://www.gizmo.com"
                    shortDesc="disabled goMenuItem" id="gmi2"/>
  <af:spacer id="stretch1" clientComponent="true"/>
  <af:commandMenuItem textAndAccessKey="Component G&uide"
                    action="guide" id="cmi9"/>
</af:menuBar>
```

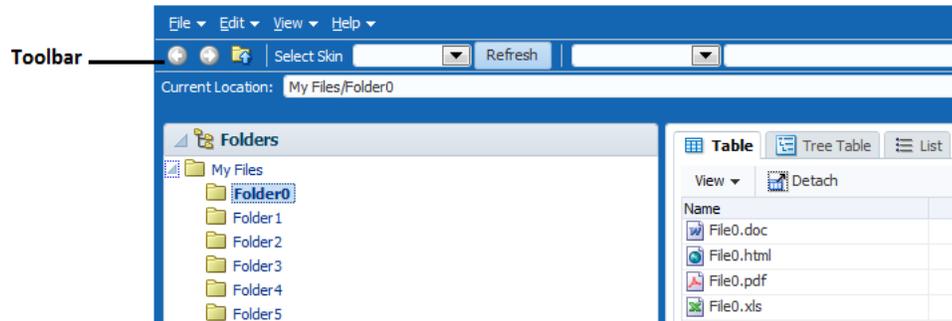
Using Toolbars

ADF Faces provides a `toolbar` component that can have icons, buttons, `inputText`, and other components. You can add components to the toolbar based on the type of operation that you want the users to perform on the application.

Along with menus, you can create toolbars in your application that contain toolbar buttons used to initiate some operation in the application. The buttons can display text, an icon, or a combination of both. Toolbar buttons can also open menus in a popup window. Along with toolbar buttons, other UI components, such as dropdown lists, can be displayed in toolbars. [Figure 15-18](#) shows the toolbar from the File Explorer application.

Tip:

Toolbars can contain buttons and links instead of toolbar buttons. However, toolbar buttons provide additional functionality, such as opening popup menus. Toolbar buttons can also be used outside of a `toolbar` component.

Figure 15-18 Toolbar in the File Explorer Application

The toolbar component can contain many different types of components, such as `inputText` components, LOV components, selection list components, and command components. ADF Faces includes a `button` component that has a `popup` facet, allowing you to provide popup menus from a toolbar button. You can configure your toolbar button so that it only opens the popup dialog and does not fire an action event. As with menus, you can group related toolbar buttons on the toolbar using the `group` component.

You can use more than one toolbar by enclosing them in a `toolbox`. Enclosing toolbars in a `toolbox` stacks them so that the first toolbar on the page is displayed at the top, and the last toolbar is displayed on the bottom. For example, in the File Explorer application, the currently selected folder name is displayed in the Current Location toolbar, as shown in [Figure 15-18](#). When you use more than one toolbar, you can set the `flex` attribute on the toolbars to determine which toolbar should take up the most space. In this case, the Current Location toolbar is set to be the longest.

If you wish toolbars to be displayed next to each other (rather than stacked), you can enclose them in a `group` component.

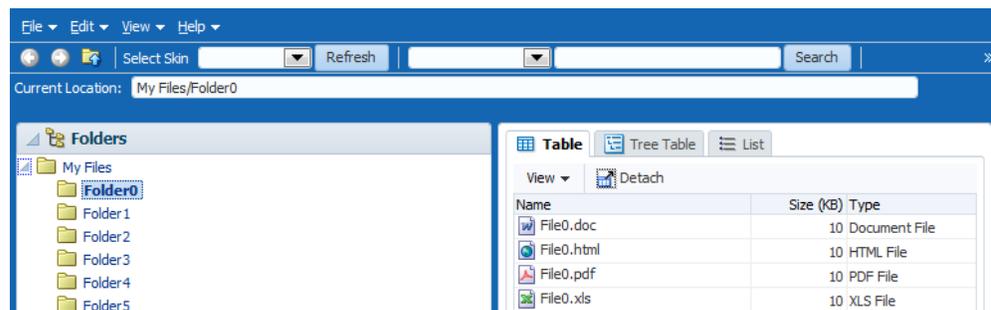
Tip:

You can also use the `toolbox` component to group menu bars with toolbars, or to group multiple menu bars. As with grouping toolbars, use the `group` component to group menu bars and toolbars on the same row.

Within a toolbar, you can set one component to stretch so that the toolbar will always be the same size as its parent container. For example, in the File Explorer application, the lower toolbar that displays the current location contains the component that shows the selected folder. This component is set to stretch so that when the window is

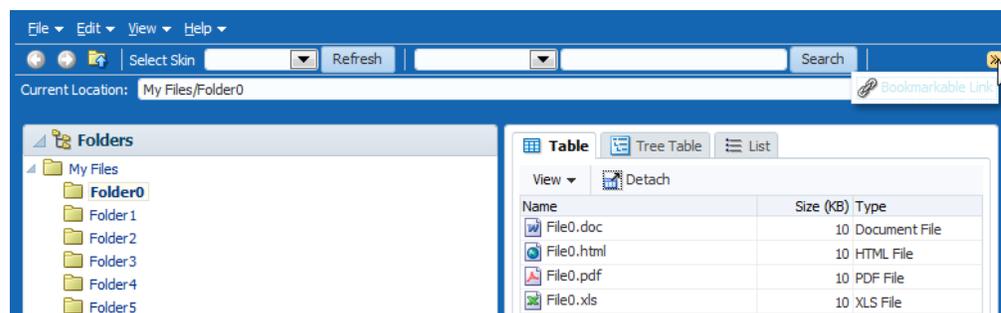
resized, that component and the toolbar will always be the same width as the parent. However, because no component in the top toolbar is set to stretch, it does not change size when the window is resized. When a window is resized such that all the components within the toolbar can no longer be displayed, the toolbar displays an overflow icon, identified by an arrow cursor in the upper right-hand corner, as shown in Figure 15-19.

Figure 15-19 Overflow Icon in a Toolbar



Clicking that overflow icon displays the remaining components in a popup window, as shown in Figure 15-20.

Figure 15-20 Toolbar Component in an Overflow Popup Window



When you expect overflow to occur in your toolbar, it is best to wrap it in a toolbox that has special layout logic to help in the overflow.

How to Create and Use Toolbars

If you are going to use more than one `toolbar` component on a page, or if you plan to use menu bars with toolbars, you first create the `toolbox` component to hold them. You then create the toolbars, and last, you create the toolbar buttons.

Tip:

If you encounter layout issues with single toolbars or menu bars, consider wrapping them in a `toolbox` component, because this component can handle overflow and layout issues.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Toolbars](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Menu and Toolbar Components](#).

To create and use toolbars:

1. If you plan on using more than one toolbar or a combination of toolbars and menu bars, create a `toolbox` component by dragging and dropping a **Toolbox** component from the Menus and Toolbars panel of the Components window.

 **Tip:**

The `panelHeader`, `showDetailHeader`, and `showDetailItem` components support a `toolbar` facet for adding toolboxes and toolbars to section headers and accordion panel headers.

2. Drag and drop a **Toolbar** onto the JSF page. If you are using a `toolbox` component, the **Toolbar** should be dropped as a direct child of the `toolbox` component.

 **Tip:**

Toolboxes also allow you to use the `iterator`, `switcher`, and `group` components as direct children, providing these components wrap child components that would usually be direct children of the toolbox.

3. If grouping more than one toolbar within a toolbox, for each toolbar, select the toolbar, expand the **Appearance** section and set the Flex attributes to determine the relative sizes of each of the toolbars. The higher the number given for the `flex` attribute, the longer the toolbox will be. For the set of toolbars shown in the following example, `toolbar2` will be the longest, `toolbar4` will be the next longest, and because their `flex` attributes are not set, the remaining toolbars will be the same size and shorter than `toolbar4`.

```
<af:toolbox>
  <af:toolbar id="toolbar1" flex="0">
    <af:button text="ButtonA"/>
  </af:toolbar>
  <af:toolbar id="toolbar2" flex="2">
    <af:button text="ButtonB"/>
  </af:toolbar>
  <af:toolbar id="toolbar3" flex="0">
    <af:button text="ButtonC"/>
  </af:toolbar>
  <af:toolbar id="toolbar4" flex="1">
    <af:button text="ButtonD"/>
  </af:toolbar>
</af:toolbox>
```

 **Performance Tip:**

At runtime, when available browser space is less than the space needed to display the contents of the toolbox, ADF Faces automatically displays overflow icons that enable users to select and navigate to those items that are out of view. The number of child components within a `toolbox` component, and the complexity of the children, will affect the performance of the overflow. You should set the size of the `toolbox` component to avoid overflow when possible. See [What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined](#).

 **Tip:**

You can use the `group` component to group toolbars (or menu bars and toolbars) that you want to appear on the same row. If you do not use the `group` component, the toolbars will appear on subsequent rows.

For information about how the `flex` attribute works, see [What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined](#).

4. Insert components into the toolbar as needed. To create a `button` on the toolbar, drag a **Button** from the General Controls section of the Components window and drop it as a direct child of the `toolbar` component.

 **Tip:**

You can use the `group` component to wrap related buttons on the bar. Doing so inserts a separator between the groups, as shown in [Figure 15-17](#).

Toolbars also allow you to use the `iterator` and `switcher` components as direct children, as long as these components wrap child components that would usually be direct children of the toolbar.

 **Tip:**

You can place other components, such as buttons and links, input components, and select components in a toolbar. However, they may not have the capability to stretch. For details about stretching the toolbar, see [Step 9](#).

5. For each `button` component, expand the Common section of the Properties window and set the following attributes:
 - **Text:** Enter the label for this toolbar button.

- **Type:** Specify a type for this toolbar button. When a toolbar button type is specified, an icon can be displayed when the button is clicked. Use one of the following acceptable `type` values:
 - **default:** Assigns no type to this toolbar button.
 - **check:** Toggles to the `depressedIcon` value if selected or to the default `icon` value if not selected.
 - **radio:** When used with other toolbar buttons in a group, makes the button currently clicked selected, and toggles the previously clicked button in the group to unselected.

 **Note:**

When setting the type to `radio`, you must wrap the toolbar button in a `group` tag that includes other toolbar buttons whose types are set to `radio` as well.

- **Selected:** Set to `true` to have this toolbar button appear as selected. The `selected` attribute is supported for checkmark- and radio-type toolbar buttons only.
- **Icon:** Set to the URI of the image file if you want to render an icon inside the component. If you render an icon, you can also set values for `hoverIcon`, `disabledIcon`, `depressedIcon`, and `iconPosition` in the **Appearance** section.

 **Tip:**

You can use either the `text` attribute (or `textAndAccessKey` attribute) or the `icon` attribute, or both.

- **IconPosition:** If you specified an icon, you can determine the position of the icon relative to the text by selecting a value from the dropdown list:
 - **<default> (leading):** Renders the icon before the text.
 - **trailing:** Renders the icon after the text.
 - **top:** Renders the icon above the text.
 - **bottom:** Renders the icon below the text.
- **Action:** Use an EL expression that evaluates to an action method in an object (such as a managed bean) that will be invoked when a user presses this button. The expression must evaluate to a public method that takes no parameters, and returns a `java.lang.Object` object.

If you want to cause navigation in response to the action generated by the button, instead of entering an EL expression, enter a static action outcome value as the value for the `action` attribute. You then must set either `partialSubmit` to `false`, or use a `redirect`. For information about configuring navigation, see [Defining Page Flows](#).

- **ActionListener:** Specify the expression that refers to an action listener method that will be notified when a user presses this button. This method can be used instead of a method bound to the `action` attribute, allowing the

`action` attribute to handle navigation only. The expression must evaluate to a public method that takes an `ActionEvent` parameter, with a return type of `void`.

6. Expand the **Appearance** section and set the following properties:
 - **HoverIcon**: Use the dropdown list to select the icon to display when the mouse cursor is directly on top of this toolbar button. If the icon is not in this menu, use the dropdown menu to the right of the list to choose **Edit**, and browse to select the icon.
 - **DepressedIcon**: Use the dropdown list to select the icon to display when the toolbar button is activated. If the icon is not in this menu, use the dropdown menu to the right of the list to choose **Edit**, and browse to select the icon.
7. Expand the **Behavior** section and set **ActionDelivery** to `none` if you do not want to fire an action event when the button is clicked. This is useful if you want the button to simply open a popup window. If set to `none`, you must have a `popup` component in the `popup` facet of the toolbar button (see Step 8), and you cannot have any value set for the `action` or `actionListener` attributes. Set to `clientServer` attribute if you want the button to fire an action event as a standard command component.
8. To have a toolbar button invoke a popup menu, insert a `menu` component into the `popup` facet of the `button` component. For information, see [How to Create and Use Menus in a Menu Bar](#).
9. If you want a toolbar to stretch so that it equals the width of the containing parent component, set **stretchId** to be the ID of the component within the toolbar that should be stretched. This one component will stretch, while the rest of the components in the toolbar remain a static size.

For example, in the File Explorer application, the `inputText` component that displays the selected folder's name is the one that should stretch, while the `outputText` component that displays the words "Current Folder" remains a static size, as shown in the following example.

```
<af:toolbar id="headerToolbar2" flex="2" stretchId="pathDisplay">
  <af:outputText id="currLocation" noWrap="true"
    value="#{explorerBundle['menuitem.location']}" />
  <af:inputText id="pathDisplay" simple="true" inlineStyle="width:100%"
    contentStyle="width:100%"
    binding="#{explorer.headerManager.pathDisplay}"
    value="#{explorer.headerManager.displayedDirectory}"
    ="true"
    validator="#{explorer.headerManager.validatePathDisplay}" />
</af:toolbar>
```

You can also use the `stretchId` attribute to justify components to the left and right by inserting a `spacer` component, and setting that component ID as the `stretchId` for the toolbar, as shown in the following example.

```
<af:toolbar flex="1" stretchId="stretch1">
  <af:button text="Forward"
    icon="/images/fwdarrow_gray.gif"
    disabled="true"></af:button>
  <af:button icon="/images/uplevel.gif" />

  <!-- Insert a stretched spacer to push subsequent buttons to the right -->

  <af:spacer id="stretch1" clientComponent="true" />
```

```
<af:button text="Reports" />
<af:button id="toggleRefresh"
           text="Refresh:OFF" />
</af:toolbar>
```

What Happens at Runtime: How the Size of Menu Bars and Toolbars Is Determined

When a page with a menu bar or toolbar is first displayed or resized, the space needed for each bar is based on the value of the bar's `flex` attribute. The percentage of size allocated to each bar is determined by dividing its `flex` attribute value by the sum of all the `flex` attribute values. For example, say you have three toolbars in a toolbox, and those toolbars are grouped together to display on the same line. The first toolbar is given a `flex` attribute value of 1, the second toolbar also has a `flex` attribute value of 1, and the third has a `flex` attribute value of 2, giving a total of 4 for all `flex` attribute values. In this example, the toolbars would have the following allocation percentages:

- Toolbar 1: $1/4 = 25\%$
- Toolbar 2: $1/4 = 25\%$
- Toolbar 3: $2/4 = 50\%$

Once the allocation for the bars is determined, and the size set accordingly, each element within the toolbars are placed left to right. Any components that do not fit are placed into the overflow list for the bar, keeping the same order as they would have if displayed, but from top to bottom instead of left to right.



Note:

If the application is configured to read right to left, the toolbars will be placed right to left. See [Language Reading Direction](#).

What You May Need to Know About Toolbars

Toolbars are supported and rendered by parent components such as `panelHeader`, `showDetailHeader`, and `showDetailItem`, which have a `toolbar` facet for adding toolbars and toolbar buttons to section headers and accordion panel headers.

Note the following points about toolbars at runtime:

- A toolbar and its buttons do not display on a header if that header is in a collapsed state. The toolbar displays only when the header is in an expanded state.
- When the available space on a header is less than the space needed by a toolbar and all its buttons, ADF Faces automatically renders overflow icons that allow users to select hidden buttons from an overflow list.
- Toolbars do not render on printable pages.

Accessing Keyboard in Toolbar

The arrow keys are used to move focus between toolbar items. The toolbar is a single tabstop with the left and right arrow keys being used to access the items on the toolbar. For items in overflow, the up and down arrow keys also work to move focus.

You can use the arrow key when you reach the toolbar's last (or first) component. The arrow key will move its focus to the next (or previous) component outside the toolbar and will not circle around the components.

16

Using Popup Dialogs, Menus, and Windows

This chapter describes how to create and use popups in secondary windows including dialogs, menus, and windows on JSF pages. Available options include the ability to declaratively or programmatically invoke a popup, display contextual information, reset input fields and control the automatic cancellation of inline popups.

This chapter includes the following sections:

- [About Popup Dialogs, Menus, and Windows](#)
- [Declaratively Creating Popups](#)
- [Declaratively Invoking a Popup](#)
- [Programmatically Invoking a Popup](#)
- [Displaying Contextual Information in Popups](#)
- [Controlling the Automatic Cancellation of Inline Popups](#)
- [Resetting Input Fields in a Popup](#)

About Popup Dialogs, Menus, and Windows

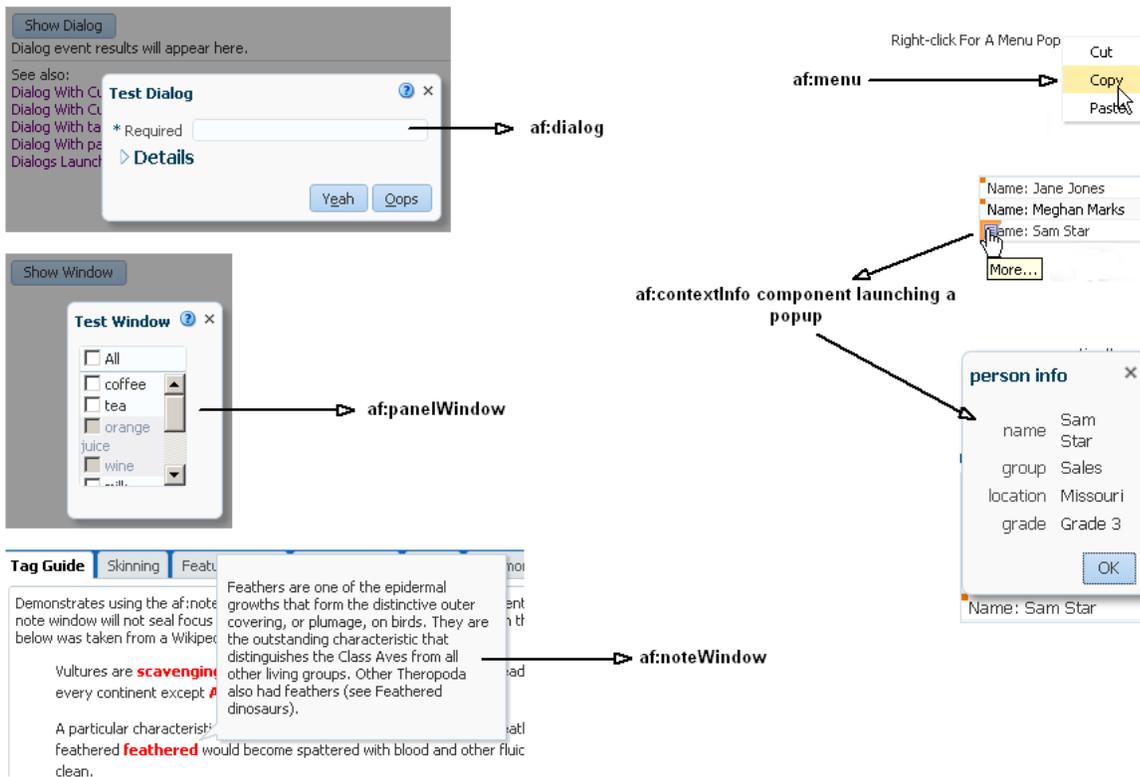
The ADF Faces `popup` component provides a popup window as a secondary window. Using the `popup` component along with the other components, you can allow users to submit input values or provide additional information to them.

You can use the `popup` component with a number of other ADF Faces components to create a variety of dialogs, menus, and windows that provide information or request input from end users. Using these components, you can configure functionality to allow your end users to show and hide information in secondary windows, input additional data, or invoke functionality. The capabilities offered by these components allow you to render content or functionality that is supplemental to the content rendered on the primary interface and, as a result, develop uncluttered and user friendly interfaces.

The `popup` component is an invisible layout control, used in conjunction with other components to display inline (that is, belonging to the same page) dialogs, windows, and menus. The `popup` component is invoked from within the primary interface and the application manages the content that renders in the `popup` component like content in the primary interface without interference from popup blockers. It is recommended that the content type you render in a `popup` component be HTML. Other types of content, such as Flash or PDF files, may not render appropriately in a `popup` component.

[Figure 16-1](#) shows examples where the `popup` component works with other ADF Faces components to render secondary windows.

Figure 16-1 ADF Faces Components for Dialogs, Menus, and Windows



To provide support for building pages for a process displayed *separate* from the parent page, ADF Faces provides a **dialog framework**. This framework supports multiple dialog pages with a control flow of their own. For example, say a user is checking out of a website after selecting a purchase and decides to sign up for a new credit card before completing the checkout. The credit card transaction could be launched using the dialog framework in an external browser window. The completion of the credit card transaction does not close the checkout transaction on the original page.

This dialog framework can also be used inline as part of the parent page. This can be useful when you want the pages to have a control flow of their own, but you do not want the external window blocked by popup blockers.

If your application uses the full Fusion technology stack, note that this dialog framework is integrated with ADF Controller for use with ADF task flows. See *Running a Bounded Task Flow in a Modal Dialog* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Using a context parameter named `LAST_WINDOW_SESSION_TIMEOUT` in your application's `web.xml` file, you can specify the maximum inactive period of time before session timeout when an application has only one open window. The maximum inactive period of time that you specify for the context parameter should be less than the value you specify for session timeout. If you enable this feature and there is only one window open in a session, the session timeout is set to the value that you specify for this context parameter. The following example shows how to set the value of the `LAST_WINDOW_SESSION_TIMEOUT` context parameter in a `web.xml` file to 1800 seconds.

For more information about configuring your application's `web.xml` file, see [Configuration in web.xml](#).

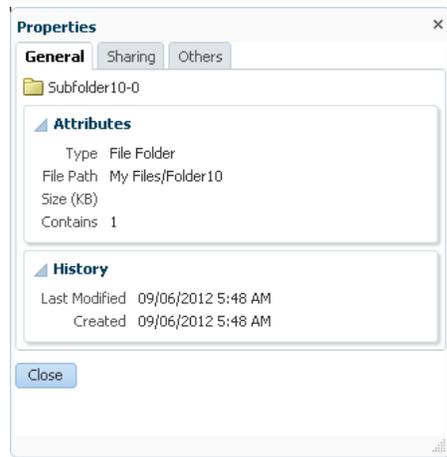
```
<!-- Sets the session timeout to 1800 seconds when there is only one window open
in the session and 1800 seconds is smaller then the original session timeout. This
gives your application the option to end the session when an end user closes the
last window. Specify a value in seconds. A negative value disables this feature.
The default value is -1. -->
<context-param>
  <param-name>LAST_WINDOW_SESSION_TIMEOUT</param-name>
  <param-value>1800</param-value>
</context-param>
```

You can also configure your application by setting the context parameter using Java API. See [What You May Need to Know About ADF Faces Window Manager Configuration](#).

Popup Dialogs, Menus, Windows Use Cases and Examples

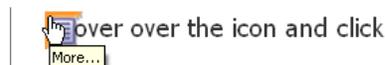
You can place a `dialog` component as a child to a `popup` component and render a dialog in a popup at runtime. The `dialog` component must be the only immediate child component of the `popup` component. At runtime, end users can view or enter information (for example, search criteria) and use the `dialog` component's default buttons to invoke a `dialogEvent` when clicked. [Figure 16-2](#) shows an example where an end user can dismiss the dialog by clicking the **Close** button.

Figure 16-2 `af:dialog` Component



You can also use components within a popup to display contextual information related to another component. When so configured, the related component displays a small square. When moused over, the icon grows and also displays a note icon, as shown in [Figure 16-3](#).

Figure 16-3 With Mouseover, Larger Icon with Note Is Displayed



When the user clicks the note icon, the associated popup displays its enclosed content.

Additional Functionality for Popup Dialogs, Menus, and Windows

You may find it helpful to understand other ADF Faces features before you use a `popup` component to create dialogs, menus, and windows. Additionally, once you have added a `popup` component (or related components) to your page, you may find that you need to add functionality such as accessibility and localization. Following are links to other functionality that these components can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want the text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Events:** The `dialog` component renders ADF Faces `button` components. You can also use a `button` component in conjunction with the `showPopupBehavior` tag to launch a popup. The `button` component used in conjunction with the `showPopupBehavior` tag delivers `ActionEvent` events when activated. For information about how to handle events on the server as well as on the client, see [Handling Events](#).
- **Messages:** Popup dialogs and secondary windows are frequently used to provide different levels of help information for users. For information about how to display messages to users, see [Displaying Tips, Messages, and Help](#).
- **Localization:** Instead of directly entering text for labels in the popup dialogs, menus, and windows that you create, you can use property files. These files allow you to manage translation of the text strings. See [Internationalizing and Localizing Pages](#).
- **Skins:** You can change the look and feel of the components that you use to create popup dialogs, menus, and windows by changing the skin. See [Customizing the Appearance Using Styles and Skins](#).
- **Accessibility:** You can make your popup dialogs, menus, and windows accessible. See [Developing Accessible ADF Faces Pages](#).
- **Dialog framework:** If your application uses the full Fusion technology stack, note that the dialog framework is integrated with ADF Controller for use with ADF task flows. See *Using Dialogs in Your Application in [Developing Fusion Web Applications with Oracle Application Development Framework](#)*.
- **Touch Devices:** ADF Faces components may behave and display differently on touch devices. See [Creating Web Applications for Touch Devices Using ADF Faces](#).
- **Drag and Drop:** You can configure your components so that the user can drag and drop them to another area on the page. See [Adding Drag and Drop Functionality](#).

What You May Need to Know About ADF Faces Window Manager Configuration

ADF Faces window lifecycle states are managed by the window manager. This is in agreement with page flow control, particularly for the management of bounded taskflows. To configure the ADF Faces window manager, you use `<context-param>`

elements in the `web.xml` file. However, for applications that insulate page developers from the `web.xml` file, you can configure the ADF Faces window manager by using the Java API, which is similar to J2EE web application session management.

You can use the `oracle.adf.view.rich.context.WindowManagerContext` to define the programmatic interface for setting the window manager configuration. The following list indicates the examples of Java API calls to configure the context for the ADF Faces window manager:

- To specify an external context:

```
extContext = FacesContext.getCurrentInstance().getExternalContext();
```

- To specify a request context:

```
rc = RequestContext.getCurrentInstance();
```

- To specify a window manager context:

```
wmContext = new WindowManagerContext(rc.getWindowManager());  
wmContext.setMaxWindowCacheSize(extContext, 100);
```

The following table lists the context parameters that you use in the `web.xml` and their Java API equivalent:

web.xml context-param	Java API	Description
LAST_WINDOW_SESSION_TIMEOUT	<pre>getLastWindowSessionTimeoutInterval setLastWindowSessionTimeoutInterval</pre>	<p>Specifies the session timeout in seconds applied when the last window moves to an unloaded state.</p> <p>Unloaded windows represent closed browser windows or browser windows that have navigated away from the application.</p> <p>The value will be applied to the <code>setMaxInactiveInterval</code> of the active session. The default value is <code>-1</code>, indicating the feature is disabled.</p>
UNLOADED_WINDOW_CLOSE_TIMEOUT	<pre>getUnloadedWindowCloseTimeoutInterval setUnloadedWindowCloseTimeoutInterval</pre>	<p>Specifies the number of seconds before an unloaded window will implicitly transition to a closed state.</p> <p>Moving the window to the closed state releases associated resources cached in session state.</p> <p>The default is 300 seconds (5 minutes). Setting the value equal to the web application session timeout disables the feature.</p>

web.xml context-param	Java API	Description
MAX_WINDOW_CACHE_SIZE	getMaxWindowCacheSize setMaxWindowCacheSize	Specifies the maximum number of windows that can be open in a session cache. Once the threshold has been reached, the most infrequently used window will implicitly move to a closed state for the most recent window open request. The default is 1000.

 **Note:**

When a window transitions to a closed state, all sub-session resources associated with that window are released. If the browser window reenters into a bounded taskflow, where the window was previously closed, ADF Controller treats the request similar to a session timeout and redirects to a safe reentry point, such as the home page.

Declaratively Creating Popups

ADF Faces provides a `showPopupBehavior` tag that allows you to declaratively create popup along with action events on buttons in the components. You can use different components in a popup component.

The `dialog`, `panelWindow`, `menu`, and `noteWindow` components can all be used inside the `popup` component to display inline popups, as shown in [Table 16-1](#). When no child component exists for the `popup` component, a very simple inline popup appears.

Table 16-1 Components Used with the popup Component

Component	Displays at Runtime
<code>dialog</code>	Displays its children inside a dialog and delivers events when the OK, Yes, No, and Cancel actions are activated. See How to Create a Dialog .

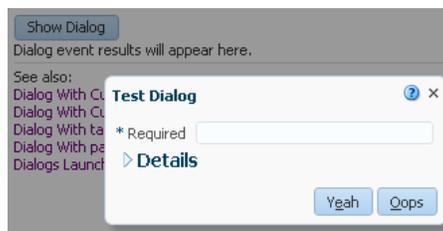
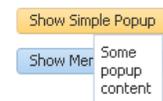
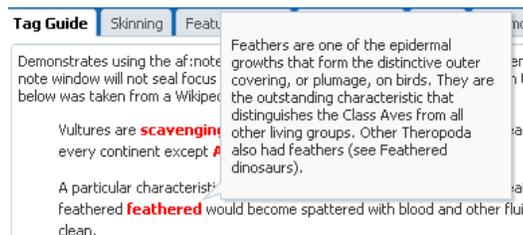
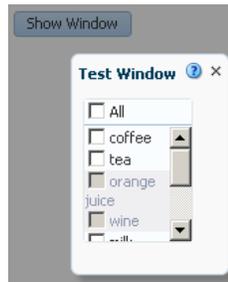


Table 16-1 (Cont.) Components Used with the popup Component

Component	Displays at Runtime
panelWindow	Displays its children in a window that is similar to a dialog, but does not support events. See How to Create a Panel Window .
menu	Displays a context menu for an associated component. See How to Create a Context Menu .
noteWindow	Displays read-only information associated with a particular UI component. Note windows are used to display help and messages and are commonly shown on mouseover or on focus gestures. See How to Create a Note Window .
popup component without one of the following components as an immediate child component: dialog, panelWindow, menu, or noteWindow	Displays content inline.



Both the `dialog` and `panelWindow` components support definition help, content displayed when a user moves the cursor over a help icon (a blue circle with a question mark). The `dialog` and `panelWindow` components do not support instruction help. See [Displaying Tips, Messages, and Help](#).

Typically, you use a `button` component in conjunction with the `showPopupBehavior` tag to launch a popup. You associate the `showPopupBehavior` tag with the component it should launch. This tag also controls the positioning of the popup (when needed).

In addition to being used with action events on `button` components, the `showPopupBehavior` tag can be used with other events, such as the `showDetail` event and the `selection` event. See [Declaratively Invoking a Popup](#).

As an alternative to using the `showPopupBehavior` tag with an action component, you can launch, cancel, or hide a popup by writing a backing bean method. The backing bean method you write takes the `actionEvent` returned by the action component as an argument. For more information about this alternative, see [Programmatically Invoking a Popup](#).

By default, the content of the popup is not sent from the server until the popup is displayed. This represents a trade-off between the speed of showing the popup when it is opened and the speed of rendering the parent page. Once the popup is loaded, by default the content will be cached on the client for rapid display.

You can modify this content delivery strategy by setting the `contentDelivery` attribute on the `popup` component to one of the following options:

- `lazy` - The default strategy previously described. The content is not loaded until you show the popup once, after which it is cached.
- `immediate` - The content is loaded onto the page immediately, allowing the content to be displayed as rapidly as possible. Use this strategy for popups that are consistently used by all users every time they use the page.
- `lazyUncached` - The content is not loaded until the popup displays, and then the content reloads every time you show the popup. Use this strategy if the popup shows data that can become stale or outdated.

If you choose to set the `popup` component's `contentDelivery` attribute to `lazy`, you can further optimize the performance of the `popup` component and the page that hosts it by setting another `popup` component attribute (`childCreation`) to `deferred`. This defers the creation of the `popup` component's child components until the application delivers the content. The default value for the `childCreation` attribute is `immediate`.

How to Create a Dialog

Create a dialog when you need the dialog to raise events when dismissed. Once you add the `dialog` component as a child to the `popup` component, you can add other components to display and collect data.

By default, the `dialog` component can have the following combination of buttons:

- Cancel
- OK
- OK and Cancel
- Yes and No

- Yes, No, and Cancel
- None

These buttons launch a `dialogEvent` when clicked. You can add other buttons to a dialog using the `buttonBar` facet. Any buttons that you add do not invoke the `dialogEvent`. Instead, they invoke the standard `actionEvent`. To make sure that the `actionEvent` invokes only on components within the dialog, set the `partialSubmit` attribute of any button that you add to `true`. However, you can add buttons and set their `partialSubmit` attribute to `false` if you set the `af:popup` component's `autoCancel` property's value to `disabled`. Choosing this latter option (`partialSubmit` set to `false`) results in increased wait times for end users because your application reloads the page and reinitializes components on the page before it restores the `popup` component's visibility (and by extension, the `dialog` component). Note that you must set the button's `partialSubmit` attribute to `true` if the `af:popup` component's `autoCancel` property's value is set to `enabled` (the default value). For information about the use of the `af:popup` component's `autoCancel` property, see [Controlling the Automatic Cancellation of Inline Popups](#).

Before you begin:

It may be helpful to understand how the `dialog` component's attributes and other components affect the functionality of inline dialogs. See [Declaratively Creating Popups](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To create an inline dialog:

1. In the Components window, from the Layout panel, in the Secondary Windows group, drag a **Popup** and drop it onto the page.

 **Tip:**

It does not matter where the `popup` component appears on the page, as the position is driven by the component used to invoke the `popup`. However, the `popup` component must be within a `form` component.

2. In the Properties window, expand the **Common** section and set the following attributes:
 - **ContentDelivery:** Select how the content is delivered to the component in the `popup`.

 **Tip:**

Values of input components in a dialog are not reset when a user clicks the dialog's **Cancel** button. If the user opens the dialog a second time, those values will still display. If you want the values to match the current values on the server, then set the `contentDelivery` attribute to `lazyUncached`.

- **Animate:** Configure animation for the popup dialog. See [What You May Need to Know About Animation and Popups](#).
 - **LauncherVar:** Enter a variable to be used to reference the launch component. This variable is reachable only during event delivery on the popup or its child components, and only if the **EventContext** is set to `launcher`.
 - **EventContext:** Set to `launcher` if the popup is shared by multiple objects, for example if the dialog within the popup will display information for the selected row in a table. Setting this attribute to `launcher` makes the row clicked current before the event listener is called, and returns data only for that row. See [What Happens at Runtime: Popup Component Events](#).
3. Optionally, in the Properties window, set the following attributes:
- **AutoCancel:** Select **disabled** to prevent the automatic cancellation of an inline popup. See [Controlling the Automatic Cancellation of Inline Popups](#).
 - **ChildCreation:** Select **deferred** to defer the creation of the `popup` component's child components until the application delivers the content. The default value for the `childCreation` attribute is `immediate`. See [Declaratively Creating Popups](#).
 - **ResetEditableValues:** Select **whenCanceled** to reset editable values that an end user entered to `null` if the end user cancels the dialog.

Alternatively, you can use the `resetListener` component. For information about using the `resetListener` component, see [Resetting Input Fields in a Popup](#).

4. In the Components window, drag and drop a **Dialog** as a direct child to the `popup` component.
5. In the Properties window, expand the **Common** section and set the following attributes:
- **Type:** Select the built-in partial-submit buttons you want to display in your dialog.

For example, if you set the `type` attribute to `yesNoCancel`, the dialog displays **Yes**, **No**, and **Cancel** buttons. When any of these buttons are pressed, the dialog dismisses itself, and the associated outcome (either `ok`, `yes`, `no`, or `cancel`) is delivered with an event. The `ok`, `yes`, and `no` outcomes are delivered with the `dialogEvent`. Cancel outcomes are sent with the `PopupCanceled` event. You can use the appropriate listener property to bind to a method to handle the event, using the outcome to determine the logic.

 **Tip:**

A dialog will not dismiss if there are any ADF Faces messages with a severity of error or greater.

- **Title:** Enter text to be displayed as the title on the dialog window.
- **CloselconVisible:** Select whether or not you want the **Close** icon to display in the dialog.
- **Modal:** Select whether or not you want the dialog to be modal. Modal dialogs do not allow the user to return to the main page until the dialog has been dismissed.

- **Resize:** Select whether or not you want users to be able to change the size of the dialog. The default is `off`.
- **StretchChildren:** Select whether or not you want child components to stretch to fill the dialog. When set to `first`, the dialog stretches a single child component. However, the child component must allow stretching. See [Geometry Management and Component Stretching](#).

 **Note:**

If you set **Resize** to `on` or set **StretchChildren** to `first`, you must also set **ContentWidth** and **ContentHeight** (see Step 8). Otherwise, the size will default to 250x250 pixels.

6. Expand the **Appearance** section and set the text attributes.

Instead of specifying separate button text and an access key, you can combine the two, so that the access key is part of the button text. Simply precede the letter to be used as an access key with an ampersand (&).

For example, if you want the text for the affirmative button to be **OK**, and you want the **O** in **OK** to be the access key, enter `&OK`.

7. Expand the **Behavior** section and if needed, enter a value for the **DialogListener** attribute. The value should be an EL expression method reference to a dialog listener method that handles the event.

For example, suppose you create a dialog to confirm the deletion of an item. You might then create a method on a managed bean similar to the `deleteItem` method shown in the following example. This method accesses the outcome from the event. If the outcome is anything other than `yes`, the dialog is dismissed. If the outcome is `yes` (meaning the user wants to delete the item), the method then gets the selected item and deletes it.

```
public void deleteItem(DialogEvent dialogEvent)
{
    if (dialogEvent.getOutcome() != DialogEvent.Outcome.yes)
    {
        return;
    }

    // Ask for selected item from FileExplorerBean
    FileItem selectedFileItem = _feBean.getLastSelectedFileItem();
    if (selectedFileItem == null)
    {
        return;
    }
    else
    {
        // Check if we are deleting a folder
        if (selectedFileItem.isDirectory())
        {
            _feBean.setSelectedDirectory(null);
        }
    }

    this.deleteSelectedFileItem(selectedFileItem);
}
```

The following example shows how to bind the `dialogListener` attribute to the `deleteItem` method.

```
<af:dialog title="#{explorerBundle['deletepopup.popuptitle']}"
  type="yesNo"
  dialogListener="#{explorer.headerManager.deleteItem}"
  id="dl">
```

The `dialogEvent` is propagated to the server only when the outcome is `ok`, `yes`, or `no`. You can block this if needed. See [How to Prevent Events from Propagating to the Server](#).)

If the user instead clicks the **Cancel** button (or the **Close** icon), the outcome is `cancel`, the `popupCancel` client event is raised on the `popup` component. Any values entered into input components rendered in the `popup` component do not get sent to the server. Any editable components that have changed their values since the `popup` component rendered do not send the changed values to the server. The `popupCancel` event is delivered to the server.

8. If you want to set a fixed size for the dialog, or if you have set **resize** to `on` or set **stretchChildren** to `first`, expand the **Appearance** section and set the following attributes:
 - **ContentHeight**: Enter the desired height in pixels.
 - **ContentWidth**: Enter the desired width in pixels.

 **Tip:**

While the user can change the values of these attributes at runtime (if the `resize` attribute is set to `on`), the values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see [Allowing User Customization on JSF Pages](#).

 **Note:**

If you use an action component without the `showPopupBehavior` tag to launch the dialog, and if that action component has values for the `windowHeight` and `windowWidth` attributes, the values on the action component override the `contentHeight` and `contentWidth` values. The dialog framework allows you to use an action component to launch a dialog without the `showPopupBehavior` tag. See *Running a Bounded Task Flow in a Modal Dialog* in *Developing Fusion Web Applications with Oracle Application Development Framework*. For information about the `showPopupBehavior` tag, see [Declaratively Invoking a Popup](#).

9. If needed, add `button` components to the `buttonBar` facet. The `button` components that you add invoke a standard `actionEvent` rather than a `dialogEvent`. To make sure that an `actionEvent` invokes only on components within the dialog, set the `button` component's `partialSubmit` attribute to `true`. You can set the `button` component's `partialSubmit` attribute to `false` if the `af:popup` component's `autoCancel` property is set to `disabled`. The values of an

`af:popup` component's `autoCancel` property and a `button` component's `partialSubmit` property determine how the `button` component dismisses and reloads a dialog. See [Controlling the Automatic Cancellation of Inline Popups](#).

 **Tip:**

If the facet is not visible in the visual editor, right-click the `dialog` component in the Structure window and choose **Facets - Dialog > ButtonBar**. Facets in use on the page are indicated by a checkmark in front of the facet name.

By default, added `button` components do not dismiss the dialog. You need to bind the `actionListener` on the `button` component to a handler that manages closing the dialog, as well as any needed processing. For examples on how to do this, see the tag documentation.

10. Insert components to display or collect data for the dialog. Use a layout component like `panelGroupLayout` to contain the components.

 **Tip:**

Normally, clicking a dialog's **Cancel** button or **Close** icon prevents any data entered into an `inputText` component from being submitted. However, setting the `autoSubmit` attribute to `true` on an `inputText` component in a dialog overrides the dialog's cancel behavior, as this setting causes a submit.

11. Add logic on the parent page to invoke the popup and dialog. See [Declaratively Invoking a Popup](#).

How to Create a Panel Window

The `panelWindow` component is similar to the `dialog` component, but it does not allow you to configure the buttons or to add buttons to a facet. If you need to invoke logic to handle data in the `panelWindow`, you need to create a listener for the `popup` component's `cancel` event.

The `popup` component that contains the `panelWindow` component must be contained within a `form` component.

 **Tip:**

If you are using the `panelWindow` as an inline popup in an application that uses the Fusion technology stack, and you want to emulate the look of a dialog, place the `panelWindow` component in the center facet of a `panelStretchLayout` component, and place `button` components in the bottom facet.

Before you begin:

It may be helpful to understand how the `panelWindow` component's attributes affect the functionality of inline windows. See [Declaratively Creating Popups](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To create an inline window:

1. In the Components window, from the Layout panel, in the Secondary Windows group, drag a **Popup** and drop it onto the page.

 **Tip:**

It does not matter where the `popup` component appears on the page, as the position is driven by the component used to invoke the popup. However, the `popup` component must be within a `form` component.

2. In the Properties window, expand the **Common** section and set the following attributes:
 - **ContentDelivery:** Select how the content is to be delivered to the component in the popup.

 **Tip:**

Values of input components are not reset when a user closes the `panelWindow` component. If the user opens the window a second time, those values will still display. If you want the values to match the current values on the server, then set the `contentDelivery` attribute to `lazyUncached`.

- **Animate:** Configure animation for the popup window. See [What You May Need to Know About Animation and Popups](#).
 - **LauncherVar:** Enter a name (for example, `source`) for a variable. Similar to the `var` attribute on a table, this variable is used to store reference in the Request scope to the component containing the `showPopupBehavior` tag. The variable is reachable only during event delivery on the `popup` or its child components, and only if **EventContext** is set to `launcher`.
 - **EventContext:** Select **launcher** if the popup is shared by multiple objects, for example if the window within the popup will display information for the selected row in a table. Setting this attribute to `launcher` makes the row clicked current before the event listener is called, and returns data only for that row. See [What Happens at Runtime: Popup Component Events](#).
 - **PopupCancelListener:** Set to an EL expression that evaluates to a handler with the logic that you want to invoke when the window is dismissed.
 - Optionally, select a value from the **AutoCancel** dropdown list to determine the automatic cancel behavior. See [Controlling the Automatic Cancellation of Inline Popups](#).
3. In the Components window, from the Layout panel, in the Secondary Windows group, drag and drop a **Panel Window** as a direct child to the `popup` component.

4. In the Properties window, expand the **Common** section and set the following attributes:
 - **Modal**: Select whether or not you want the window to be modal. Modal windows do not allow the user to return to the main page until the window has been dismissed.
 - **CloseIconVisible**: Select whether or not you want the **Close** icon to display in the window.
 - **Title**: The text displayed as the title in the window.
 - **Resize**: Select whether or not you want users to be able to change the size of the dialog. The default is `off`.
 - **StretchChildren**: Select whether or not you want child components to stretch to fill the window. When set to `first`, the window stretches a single child component. However, the child component must allow stretching. See [Geometry Management and Component Stretching](#).

 **Note:**

If you set **Resize** to `on` or set **StretchChildren** to `first`, you must also set **ContentWidth** and **ContentHeight** (see Step 6). Otherwise, the size will default to 250x250 pixels.

5. If you want to set a fixed size for the window, or if you have set **Resize** to `on` or set **StretchChildren** to `first`, expand the **Appearance** section and set the following attributes:
 - **ContentHeight**: Enter the desired height in pixels.
 - **ContentWidth**: Enter the desired width in pixels.

 **Tip:**

While the user can change the values of these attributes at runtime (if the `resize` attribute is set to `on`), the values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see [Allowing User Customization on JSF Pages](#).

 **Note:**

If an action component without the `showPopupBehavior` tag is used to launch the dialog, and if that action component has values for the `windowHeight` and `windowWidth` attributes, the values on the action component will override the `contentHeight` and `contentWidth` values. For information about the `showPopupBehavior` tag, see [Declaratively Invoking a Popup](#).

6. Insert components to display or collect data for the window. Use a layout component like `panelGroupLayout` to contain the components.

7. Add logic on the parent page to invoke the popup and panel window. See [Declaratively Invoking a Popup](#).

How to Create a Context Menu

You create a context menu by using menu components within the popup component. You can then invoke the context menu popup from another component, based on a given trigger. If instead, you want toolbar buttons in a toolbar to launch popup menus, then see [Using Toolbars](#).

Before you begin:

It may be helpful to understand how the `popup` component's attributes and other components affect the functionality of context menus. See [Declaratively Creating Popups](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To create an inline context menu:

1. In the Components window, from the Layout panel, in the Secondary Windows group, drag a **Popup** and drop it onto the page.

Tip:

It does not matter where the popup component appears on the page, as the position is driven by the component used to invoke the popup. However, the `popup` component must be within a `form` component.

2. In the Properties window, expand the **Common** section and set the following attributes.
 - **ContentDelivery**: Determines how the content is delivered to the component in the popup.
 - **Animate**: Configure animation for the context menu. See [What You May Need to Know About Animation and Popups](#).
 - **LauncherVar**: Enter a variable name (for example, `source`) to be used to reference the launch component. This variable is reachable only during event delivery on the `popup` or its child components, and only if the **EventContext** is set to `launcher`.
 - **EventContext**: Set to `launcher` if the popup is shared by multiple objects, for example if the menu within the popup will display information for the selected row in a table. Setting this attribute to `launcher` makes the row clicked current before the event listener is called, and returns only data for that row. See [What Happens at Runtime: Popup Component Events](#).
 - Optionally, select a value from the **AutoCancel** dropdown list to determine the automatic cancel behavior. See [Controlling the Automatic Cancellation of Inline Popups](#).
3. In the Components window, drag and drop a **Menu** as a direct child to the `popup` component, and build your menu using `commandMenuItem` components, as described in [How to Create and Use Menus in a Menu Bar](#).

 **Tip:**

Because this is a context menu, you do not need to create a menu bar or multiple menus, as documented in Steps 1 through 5 in [How to Create and Use Menus in a Menu Bar](#).

4. Add logic on the parent page to invoke the popup and context menu. See [Declaratively Invoking a Popup](#).

How to Create a Note Window

Use the `noteWindow` component to display read-only text. The `popup` component that contains the `noteWindow` component must be contained within a `form` component.

Before you begin:

It may be helpful to understand how the `noteWindow` component's attributes and other components affect functionality. See [Declaratively Creating Popups](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To create an inline window:

1. In the Components window, from the Layout panel, in the Secondary Windows group, drag a **Popup** and drop it onto the page.

 **Tip:**

It does not matter where the popup component appears on the page, as the position is driven by the component used to invoke the popup. However, the `popup` component must be within a `form` component.

2. In the Properties window, expand the **Common** section and set the following attributes.
 - **ContentDelivery**: Determines how the content is delivered to the component in the popup.
 - **Animate**: Configure animation for the note window. See [What You May Need to Know About Animation and Popups](#).
 - **LauncherVar**: Enter a variable to be used to reference the launch component. This variable is reachable only during event delivery on the `popup` or its child components, and only if the **EventContext** is set to `launcher`.
 - **EventContext**: Select **launcher** if the popup is shared by multiple objects, for example if the window within the popup will display information for the selected row in a table. Setting this attribute to `launcher` makes the row clicked current before the event listener is called, and returns only data for that row. See [What Happens at Runtime: Popup Component Events](#).
 - **PopupCancelListener**: Set to an EL expression that evaluates to a handler with the logic that you want to invoke when the window is dismissed.

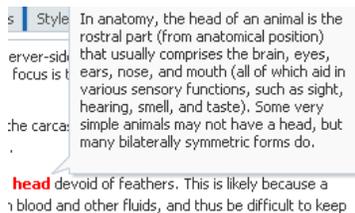
- Optionally, select a value from the **AutoCancel** dropdown list to determine the automatic cancel behavior. See [Controlling the Automatic Cancellation of Inline Popups](#).
3. In the Components window, from the Layout panel, in the Secondary Windows group, drag and drop a **Note Window** as a direct child to the `popup` component.
 4. To enter the text to display in the window:
 - a. Click the **Source** tab to view the page source code.
 - b. Remove the closing slash (`/`) from the `af:noteWindow` tag.
 - c. Below the `af:noteWindow` tag, enter the text to display, using simple HTML tags, and ending with a closed `af:noteWindow` tag.

The following example shows text for a note window.

```
<af:popup id="popupHead" contentDelivery="lazyUncached">
  <af:noteWindow inlineStyle="width:200px" id="nw3">
    <p>In anatomy, the head of an animal is the rostral part (from
      anatomical position) that usually comprises the brain, eyes,
      ears, nose, and mouth (all of which aid in various sensory
      functions, such as sight, hearing, smell, and taste). Some very
      simple animals may not have a head, but many bilaterally
      symmetric forms do.</p>
  </af:noteWindow>
</af:popup>
```

Figure 16-4 shows how the note would display.

Figure 16-4 Text Displayed in a Note Window



5. Optionally, in the Properties window, expand the **Behavior** section and specify a number of seconds in the **AutoDismissalTimeout** field. The value you specify determines the time in seconds that the note window displays before the application automatically dismisses it. Any value you specify overrides the default automatic dismissal behavior. This override is revoked if the end user moves the mouse over the content of the note window because this gesture reverts the automatic dismissal behavior back to the default automatic dismissal behavior for the note window. The default automatic dismissal behavior is to dismiss the note window when focus changes from the launching source or from the content of the popup.

 **Note:**

The feature enabled by this property is not accessible friendly because a mouse over triggers the timeout cancellation period and there is no keyboard equivalent.

6. Add logic on the parent page to invoke the popup and note window. See [Declaratively Invoking a Popup](#).

What Happens at Runtime: Popup Component Events

When content is delivered to the popup, and the `contentDelivery` attribute is set to either `lazy` or `lazyUncached`, the `popupFetch` server-side event is invoked. This event has two properties, `eventContext` and `launcherVar`. The `eventContext` property determines the context from which the event is delivered, either from the context of the popup (`self`) or from the component that launched the popup (`launcher`). Setting the context to `launcher` can be very useful if the popup is shared by multiple components, because the framework will behave as though the component that launched the popup had launched the event, and not the popup. The `launcherVar` property is used to keep track of the current launcher, similar to the way in which variables are used to stamp out rows in a table.

For example, say you have a column in a table that displays a person's first name using a `link` component. When the `link` component is hovered over, a popup `noteWindow` is invoked that shows the person's full name. Because this `noteWindow` will be used by all rows in the table, but it needs to display the full name only for the row containing the `link` component that was clicked, you need to use the `eventContext` property to make sure that the context is that row, as shown in the following example.

```
<af:popup id="noteWindow" contentDelivery="lazyUncached" eventContext="launcher"
  launcherVar="source">
  <af:noteWindow>
    <af:outputText value="#{testBean.fullName}"/>
  </af:noteWindow>
</af:popup>
<af:table var="person" value="#{testBean.people}">
  <af:column id="firstName">
    <af:link text="#{person.firstName}">
      <af:showPopupBehavior popupId="::noteWindow" triggerType="mouseHover"/>
    </af:link>
  </af:column>
</af:table>
```

Using the variable `source`, you can take values from the source and apply them, or you can set values. For example, you could get the full name value of the `people` object used in the table, and set it as the value of the `testBean`'s `fullName` property used by the window, using a `setPropertyListener` and `clientAttribute` tag, as shown in the following example.

```
<af:popup id="noteWindow" contentDelivery="lazyUncached" eventContext="launcher"
  launcherVar="source">
  <af:noteWindow>
    <af:outputText value="#{testBean.fullName}"/>
  </af:noteWindow>
  <af:setPropertyListener from="#{source.attributes.fullName}"
    to="#{testBean.fullName}" type="popupFetch"/>
</af:popup>
<af:table var="person" value="#{testBean.people}">
  <af:column id="firstName">
    <f:facet name="header">
      <af:outputText value="First Name"/>
    </f:facet>
    <af:link text="#{person.firstName}">
      <af:showPopupBehavior popupId="::noteWindow" triggerType="mouseHover"/>
    </af:link>
  </af:column>
</af:table>
```

```
        <af:clientAttribute name="fullName" value="#{person.fullName}"/>
    </af:link>
</af:column>
</af:table>
```

In this example, the `launcherVar` property source gets the full name for the current row using the `popupFetch` event. For information about using the `setPropertyListener` tag, see [How to Use the `pageFlowScope` Scope Without Writing Java Code](#). For information about using client attributes, see [Using Bonus Attributes for Client-Side Components](#). For information about the `showPopupBehavior` tag, see [Declaratively Invoking a Popup](#).

Popups also invoke the following client-side events:

- `popupOpening`: Fired when the popup is invoked. If this event is canceled in a client-side listener, the popup will not be shown.
- `popupOpened`: Fired after the popup becomes visible. One example for using this event would be to create custom rules for overriding default focus within the popup.
- `popupCanceled`: Fired when a popup is unexpectedly dismissed by auto-dismissal or by explicitly invoking the popup client component's `cancel` method. This client-side event also has a server-side counterpart.
- `popupClosed`: Fired when the popup is hidden or when the popup is unexpectedly dismissed. This client-side event also has a server-side counterpart.

When a popup is closed by an affirmative condition, for example, when the **Yes** button is clicked, it is hidden. When a popup is closed by auto-dismissal, for example when either the **Close** icon or the **Cancel** button is clicked, it is canceled. Both types of dismissals result in raising a `popupClosed` client-side event. Canceling a popup also raises a client-side `popupCanceled` event that has an associated server-side counterpart. The event will not be propagated to the server unless there are registered listeners for the event. If it is propagated, it prevents processing of any child components to the popup, meaning any submitted values and validation are ignored. You can create a listener for the `popupCanceled` event that contains logic to handle any processing needed when the popup is canceled.

If you want to invoke some logic based on a client-side event, you can create a custom client listener method. See [Listening for Client Events](#). If you want to invoke server-side logic based on a client event, you can add a `serverListener` tag that will invoke that logic. See [Sending Custom Events from the Client to the Server](#).

What You May Need to Know About Dialog Events

The `dialog` component raises a `dialogEvent` when the end user clicks the **OK**, **Yes**, **No** or **Cancel** buttons. A `dialog` component automatically hides itself when the end user clicks the **OK**, **Yes** or **No** buttons provided that no message with a severity of error or greater exists on the page. An end user selecting the **Cancel** button or close icon cancels the parent `popup` component and raises a `popup canceled` event.

You can configure a `dialogListener` attribute to intercept the `dialogEvent` returned by the **OK**, **Yes**, **No**, and **Cancel** buttons. Only the `dialogEvent` returned by the **OK**, **Yes** and **No** buttons get propagated to the server. The `dialogEvent` returned by the **Cancel** button, the ESC key, and close icon queue a client dialog event and do not get propagated to the server.

If you configure an `actionListener` for the action component that invokes a `dialog` component to carry out an action (for example, update an `inputText` component) after the `dialog` component returns, you also need to call `resetValue()` on the `inputText` component if the action component's `immediate` value is set to `true`.

For information about the events raised by the `dialog` and `popup` components, see the *Tag Reference for Oracle ADF Faces*.

What You May Need to Know About Animation and Popups

The `dialog`, `panelWindow`, `menu`, and `noteWindow` components that render inside the `popup` component to display popups can use animation when rendering. You enable animation for these components in an application by setting the `<animation-enabled>` element to `true` in the application's `trinidad-config.xml` file, as described in [Animation Enabled](#).

You can further customize or disable the animation of these components by writing values for the `-tr-animate` and `-tr-open-animation-duration` ADF skin properties in the application's ADF skin. The following example demonstrates how you can permit animation for `menu` components in an application while you disable animation for `dialog` components.

```
/** Animate menu components and specify 10 seconds as the duration to open a menu or
a popup menu */
af|menu {
    -tr-open-animation-duration: 10000;
    -tr-animate: true;
}

/** Disable animation for dialog components */
af|dialog {
    -tr-animate: false;
}
```

The animation behavior of specific instances of a `popup` component can be configured by setting the appropriate value for the `popup` component's `animate` property, as described in the following list:

- **default:** The `<animation-enabled>` element in the application's `trinidad-config.xml` file and ADF skin properties in the application's ADF skin determine the animation behavior of the `popup` component.
- **false:** Turns off animation for the `popup` regardless of the animation settings that you have configured for the application in the `trinidad-config.xml` file or the application's ADF skin. For example, assume that you set this value for a `popup` component that displays a `menu` component. Animation is disabled for this specific `popup` menu despite an application enabling animation and containing the ADF skin properties for `menu` components, as shown in the previous code example..
- **true:** Overrides an `-tr-animate: false` entry in an application's ADF skin. For example, if you set the `animate` property to `true` for a `popup` component that renders a `dialog` component, the `popup` dialog uses animation when rendering despite the ADF skin properties configured for `dialog` components, as shown in the previous code example..

For information about the ADF skin properties for animation, see *Tag Reference for Oracle ADF Faces Skin Selectors*.

Controlling Display Behavior of Popups

You can allow either the users to dismiss a popup or control the behavior of a popup by modifying the properties of the ADF Faces popup component. Use the `AutoDismissalTimeout`, `Pinning`, `Stealing Focus`, and `Positioning` properties to control the popup behavior.

You can control how the popups should display, how much time a `popup` component should be visible before it is automatically dismissed, where the focus should be either in the main window or popup window, and where the `popup` component should be aligned in a screen. You can modify these behavior by modifying the properties of the components:

- **AutoDismissalTimeout** This the number of seconds that a `popup` component displays before the application automatically dismisses it. The default automatic dismissal behavior is to dismiss the component when focus changes from the launching source or from the content of the popup.

For information about specifying the time for a `popup` component, see [How to Dismiss a Popup Component Automatically](#).

 **Note:**

The **AutoDismissalTimeout** attribute in `noteWindow` takes precedence if you specify this attribute for `popup` component as well as `noteWindow`.

- **Pinning** A new pin icon is available in the header of dialog if you have specified any valid value for the **AutoDismissalTimeout** property. When you click the pin icon, the auto dismissal behavior will be revoked and will not be re-enabled until you change the value in the **Behaviour** section again.

 **Note:**

This feature is available only for `dialog` and `panelwindow` components.

- **Stealing Focus** A new property **InitialFocus** is available for which you can set the value either as **Auto** or **None**. If you specify none, the focus remains in the main page when a dialog is opened. If you specify auto, the focus changes to the dialog from main. The default value is auto.

 **Note:**

- This feature is available only for `dialog` and `panelwindow` components.
- This property is valid only for non-modal dialogs.
- You can use the `Ctrl + Alt + w` key combination to toggle focus between windows.

- **Positioning** By default, the `popup` components display in the middle of the screen. If you specify a value for the **AlignId** property, the component is placed relative to the component.

How to Dismiss a Popup Component Automatically

The application, by default, automatically dismisses a `popup` component when focus changes from the launching source or from the content of the popup. However, you can also specify the time in seconds that a `popup` component should display before the application automatically dismisses it.

Before you begin:

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To specify time for a `popup` component:

1. In the Structure window, right-click the `af:popup` component for which you want to specify time and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section and specify the number of seconds in the **AutoDismissalTimeout** property.

Any value that you specify overrides the default automatic dismissal behavior. This override is revoked if the end user moves the mouse over the content of the popup component, because this gesture reverts the automatic dismissal behavior back to the default automatic dismissal behavior for the component.

What Happens When a Popup Component is Automatically Dismissed

When an inline popup is dismissed automatically after the specified time in the **AutoDismissalTimeout** attribute, the system raises an **AdfPopupClosedEvent** client-only event. This event is not synchronized back to the server. To send any custom data back to the server, create a new listener tag as given in the example below that leverages the existing `af:serverListener` tag and bind it to the `af:clientListener` tag without any JavaScript code needed:

```
<af:popup id="test" contentDelivery="lazyUncached">
  <af:dialog title="test">
    <af:inputText id="testInput" label="test"
binding="#{mybean.testInput}"/>
  </af:dialog>
  <af:clientServerListener method="#{mybean.clientDelegateListener}"
immediate="true" triggerType="click"/>
</af:popup>
```

Declaratively Invoking a Popup

You can use the default behavior of a popup to open a `popup` component or use the ADF Faces `showPopupBehavior` tag to control its behavior.

With ADF Faces components, JavaScript is not needed to show or hide popups. The `showPopupBehavior` tag provides a declarative solution, so that you do not have to

write JavaScript to open a `popup` component or register a script with the `popup` component. For information about client behavior tags, see [Using ADF Faces Client Behavior Tags](#).

The `showPopupBehavior` tag listens for a specified event, for example the `actionEvent` on an action component, or the `disclosureEvent` on a `showDetail` component. However, the `showPopupBehavior` tag also cancels delivery of that event to the server. Therefore, if you need to invoke some server-side logic based on the event that the `showPopupBehavior` tag is listening for, then you need to use either JavaScript to launch the popup, or programmatically launch the `popup` component, as described in [Programmatically Invoking a Popup](#).

How to Declaratively Invoke a Popup Using the `af:showPopupBehavior` Tag

You use the `showPopupBehavior` tag in conjunction with the component that invokes the popup, for example a `button` component that invokes a dialog, or an `inputText` component that, when right-clicked, will invoke a context menu.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you if you want to invoke a popup component declaratively. See [Declaratively Invoking a Popup](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

You will need to complete this task:

Create the type of popup that you want to invoke declaratively, as described in [Declaratively Creating Popups](#), and create the component that invokes the popup.

To use the `showPopupBehavior` tag:

1. In the Components window, from the Operations panel, in the Behavior group, drag a **Show Popup Behavior** and drop it as a child to the component that invokes the popup.
2. In the Properties window, choose **Edit** from the context menu that appears when you click the icon that appears when you hover over the **PopupId** property field. Use the Edit Property: PopupId dialog to select the popup component to invoke and click **OK**.
3. In the Properties window, from the **TriggerType** dropdown menu, choose the trigger to invoke the popup. The default is `action` which can be used for action components. Use `contextMenu` to trigger a popup when the right-mouse is clicked. Use `mouseHover` to trigger a popup when the cursor is over the component. The popup closes when the cursor moves off the component. For a detailed list of component and mouse/keyboard events that can trigger the popup, see the documentation for the `showPopupBehavior` tag in the *Tag Reference for Oracle ADF Faces*.

 **Note:**

The event selected for the `showPopupBehavior` tag's `triggerType` attribute will not be delivered to the server. If you need to invoke server-side logic based on this event, then you must invoke the popup using either JavaScript or a custom event as documented in [Sending Custom Events from the Client to the Server](#) or invoke the popup programmatically as documented in [Programmatically Invoking a Popup](#).

4. Choose **Edit** from the context menu that appears when you click the icon that appears when you hover over the **AlignId** property field. Use the Edit Property: AlignId dialog to select the component with which you want the popup to align.
5. In the **Align** dropdown menu, choose how the popup should be positioned relative to the component selected in the previous step.

 **Note:**

The `dialog` and `panelWindow` components do not require `alignId` or `align` attributes, as the corresponding popup can be moved by the user. If you set **AlignId**, the value will be overridden by any manual drag and drop repositioning of the dialog or window. If no value is entered for **AlignId** or **Align**, then the dialog or window is opened in the center of the browser.

Additionally, if the `triggerType` attribute is set to `contextMenu`, the alignment is always based on mouse position.

What Happens When You Use `af:showPopupBehavior` Tag to Invoke a Popup

At design time, JDeveloper generates the corresponding values in the source files that you selected in the Properties window. The following example shows sample code that displays some text in the `af:popup` component with the `id` attribute "popup2" when the button "Show Popup" is clicked.

```
<af:button text="Click me" clientComponent="true" id="popupButton2">
  <showPopupBehavior popupId="popup2" alignId="popupButton2" align="afterStart"/>
</af:button>
...
<af:popup id="popup2">
  <af:panelGroupLayout layout="vertical">
    <af:outputText value="Some"/>
    <af:outputText value="popup"/>
    <af:outputText value="content"/>
  </af:panelGroupLayout>
</af:popup>
```

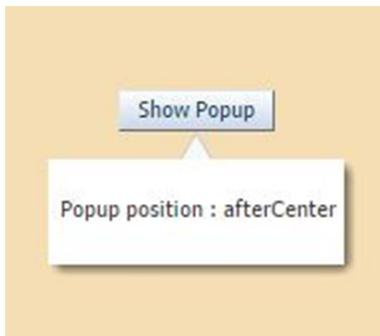
The code in the example tells ADF Faces to align the popup contents with the button identified by the `id` attribute, and to use the alignment position of `afterStart`, which aligns the popup underneath the button, as shown in [Figure 16-5](#).

Figure 16-5 Button and Popup Contents

The `tail` attribute when used with `af:showPopupBehavior` tag displays the popup window with a triangle. It has valid values `none` or `simple` or any custom value. The default value of the `tail` attribute is `none` except for the `notewindow` popup type that has the default value `simple`. The following example shows sample code that displays some text in the `af:button` component with the `id` attribute `launchElement` and `tail` attribute is set to `simple` when the button **Show Popup** is clicked.

```
<af:button id="launchElement" text="Show Popup" clientComponent="true">
  <af:showPopupBehavior popupId="popup" align="afterCenter"
  alignId="launchElement" triggerType="action" tail="simple"/>
</af:button>
```

The code in the example tells ADF Faces to align the popup contents with the button identified by the `id` attribute, and to use the alignment position of `afterCenter` which aligns the popup after the button, as shown in [Figure 16-6](#)

Figure 16-6 Popup with Tail Attribute

Programmatically Invoking a Popup

You can programmatically invoke a popup in ADF Faces components to deliver the results of an `actionEvent` to the server or you can show, hide, or cancel a popup as a result of the server-side response.

You can programmatically show, hide, or cancel a popup in response to an `actionEvent` generated by an action component. Implement this functionality if you want to deliver the `actionEvent` to the server immediately so you can invoke server-side logic and show, hide, or cancel the popup in response to the outcome of invoking the server-side logic.

Programmatically invoking a popup as described here differs to the method of invoking a popup described in [Declaratively Invoking a Popup](#) where the `showPopupBehavior` tag does not deliver the `actionEvent` to the server immediately.

You create the type of popup that you want by placing one of the components (`dialog`, `panelWindow`, `menu`, or `noteWindow`) inside the `popup` component as described in [Declaratively Creating Popups](#). Make sure that the `popup` component is in the right context when you invoke it. One of the easier ways to do this is to bind it to the backing bean for the page, as in the following example.

```
<af:popup
  id="p1"
  binding="#{mybean.popup}"
  ...
/>
```

Once you have done this, you configure an action component's `actionListener` attribute to reference the `popup` component by calling an accessor for the `popup` binding.

Write code for the backing bean method that invokes, cancels, or hides the popup. The following example shows a `showPopup` backing bean method that uses the `HINT_LAUNCH_ID` hint to identify the action component that passes the `actionEvent` to it and `p1` to reference the popup on which to invoke the `show` method.

```
public void showPopup(ActionEvent event) {
{
  FacesContext context = FacesContext.getCurrentInstance();
  UIComponent source = (UIComponent)event.getSource();
  String alignId = source.getClientId(context);
  RichPopup.PopupHints hints = new RichPopup.PopupHints();
  hints.add(RichPopup.PopupHints.HintTypes.HINT_ALIGN_ID,source)
  .add(RichPopup.PopupHints.HintTypes.HINT_LAUNCH_ID,source)
  .add(RichPopup.PopupHints.HintTypes.HINT_ALIGN,
  RichPopup.PopupHints.AlignTypes.ALIGN_AFTER_END);
  p1.show(hints);
}
}
```

[Example 16-1](#) shows a backing bean method that cancels a popup in response to an `actionEvent` while [Example 16-2](#) shows a backing bean method that hides a popup in response to an `actionEvent`. The `p1` object in the following examples refers to an instance of the `RichPopup` class from the following package:

```
oracle.adf.view.rich.component.rich.RichPopup
```

For information about `RichPopup`, see *Java API Reference for Oracle ADF Faces*.

Example 16-1 Backing Bean Method Canceling a Popup

```
public void cancelPopupActionListener(ActionEvent event) {
  FacesContext context = FacesContext.getCurrentInstance();
  p1.cancel();
}
}
```

Example 16-2 Backing Bean Method Hiding a Popup

```
public void hidePopupActionListener(ActionEvent event) {
  FacesContext context = FacesContext.getCurrentInstance();
  p1.hide();
}
}
```

How to Programmatically Invoke a Popup

You configure the action component's `actionListener` attribute to reference the backing bean method that shows, cancels or hides the popup.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you if you want to invoke a popup component programmatically. See [Programmatically Invoking a Popup](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

You will need to complete this task:

Create the type of popup that you want the server-side method to invoke, as described in [Declaratively Creating Popups](#).

To programmatically invoke a popup:

1. In the Components window, from the General Controls panel, drag and drop an action component onto the JSF page. For example, a **Button**.
2. In the Properties window, expand the **Behavior** section and set the following attributes:
 - **PartialSubmit**: Select **true** if you do not want the Fusion web application to render the entire page after an end user clicks the action component. The default value (`false`) causes the application to render the whole page after an end user invokes the action component. For information about page rendering, see [Rerendering Partial Page Content](#).
 - **ActionListener**: Set to an EL expression that evaluates to a backing bean method with the logic that you want to execute when the end user invokes the action component at runtime.
3. Write the logic for the backing bean that is invoked when the action component in Step 2 passes an `actionEvent`.

What Happens When You Programmatically Invoke a Popup

At runtime, end users can invoke the action components you configure to invoke the server-side methods to show, cancel, or hide a popup. For example, [Figure 16-7](#) shows a `panelWindow` component that renders inside a `popup` component. The `panelWindow` component exposes two buttons (**Cancel** and **Hide**) that invoke the `cancel` and `hide` methods respectively. End users invoke a `link` component rendered in the **SupplierName** column of the `table` component in the underlying page to show the popup.

Figure 16-7 Popup Component Invoked by a Server-Side Method

Suppliers ID	Supplier's Name	Supplier Status	Phone Number	Email	Creation Date
100	Stuffz	ACTIVE	402.555.0158	contact@stuffz.ex	17/03/2011
101	Nexus	ACTIVE	608.555.0114	contact@nexus.ex	17/03/2011
102	Gifts-N-More	ACTIVE	225.555.0181	contact@giftsnmor	17/03/2011
103	E				17/03/2011
104	J				17/03/2011
105	C				17/03/2011
106	T				17/03/2011
107	M				17/03/2011
108	BigSwamp	ACTIVE	248.555.0154	contact@bigswamp	17/03/2011
109	7-Mart	ACTIVE	959.555.0120	contact@7mart.ex	17/03/2011

Panel Window inside Popup Component to Change Supplier's Name

Enter a new name:

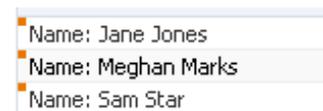
OK Cancel Hide

Displaying Contextual Information in Popups

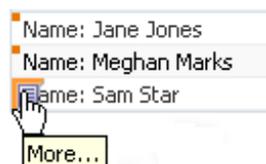
The ADF Faces `contextInfo` component allows you to display additional information to the users in a popup window. Using this component, you can display a message, a warning, a hint, and other additional information to the users when they right-click the component.

There may be cases when you think the user may need more information to complete a task on a page, but you don't want to clutter the page with information that may not be needed each time the page is accessed, or with multiple buttons that might launch dialogs to display information. While you could put the information in a popup that a user launches when they right-click a component, the user would have no way of knowing the information was available in a popup.

The `contextInfo` component allows you to display additional information in a popup and also notifies users that additional information is available. When you place the `contextInfo` component into the context facet of a component that supports contextual information, a small orange square is shown in the upper left-hand corner of the component, as shown in [Figure 16-8](#).

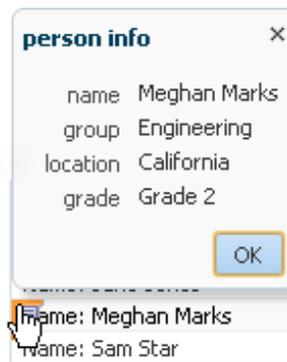
Figure 16-8 `contextInfo` Displays a Square

When the user places the cursor over the square, a larger triangle with a note icon and tooltip is displayed, indicating that additional information is available, as shown in [Figure 16-9](#).

Figure 16-9 `contextInfo` Component Indicates Additional Information Is Available

Because a `showPopupBehavior` tag is a child to the `contextInfo` component, the referenced popup displays when the user clicks the information icon, as shown in [Figure 16-10](#).

Figure 16-10 Dialog launched From `contextInfo` Component



How to Create Contextual Information

You use the `showPopupBehavior` component as a child to the `contextInfo` component, which allows the popup component to align with the component that contains the `contextInfo` component.

Before you begin:

1. Create the component that will be the parent to the `contextInfo` component. The following components support the `contextInfo` component:
 - `column`
 - `link`
 - `inputComboboxListOfValues`
 - `inputListOfValues`
 - `inputText`
 - `outputFormatted`
 - `outputText`
 - `selectOneChoice`
2. Create the popup to display, as documented in [Declaratively Creating Popups](#).
3. You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To use a `contextInfo` component:

1. In the Components window, from the General Controls panel, drag a **Context Info** and drop it into the `Context` facet of the component that is to display the additional information icons.

 **Tip:**

If the facet is not visible in the visual editor, right-click the component in the Structure window and choose **Facets - component name > Context**. Facets in use on the page are indicated by a checkmark in front of the facet name.

2. If you need server-side logic to execute when the `contextInfo` component displays, in the Properties window, expand the **Behavior** section and bind the **ContextInfoListener** to a handler that can handle the event

 **Note:**

If you use the `showPopupBehavior` tag to launch the popup, then delivery of the `contextInfoEvent` to the server is cancelled. If you need to invoke server-side logic based on this event, then you must launch the popup by using either JavaScript or a custom event as documented in [Sending Custom Events from the Client to the Server](#).

3. In the Components window, from the Operations panel, in the Behavior group, drag a **Show Popup Behavior** and drop it as a child to the `contextInfo` component.
4. With the `showPopupBehavior` tag selected in the editor, in the Properties window, set the attributes as described in [How to Declaratively Invoke a Popup Using the af:showPopupBehavior Tag](#). For the **TriggerType**, make sure to enter `contextInfo`.

Controlling the Automatic Cancellation of Inline Popups

The inline popup is automatically cancelled by the ADF web application; however, you can control this default behavior by disabling the automatic cancellation of an inline popup component.

You can use the `popup` component with a number of other components to create inline popups. That is, inline windows, dialogs, and context menus. These other components include the:

- Dialog component to create an inline dialog
See [How to Create a Dialog](#).
- `panelWindow` component to create an inline window
See [How to Create a Panel Window](#).
- Menu components to create context menus
See [How to Create a Context Menu](#).
- `noteWindow` component to create a note window
See [How to Create a Note Window](#).

By default, a Fusion web application automatically cancels an inline popup if the metadata that defines the inline popup is replaced. Scenarios where this happens include the following:

- Invocation of an action component that has its `partialSubmit` property set to `false`. The Fusion web application renders the entire page after it invokes such an action component. In contrast, an action component that has its `partialSubmit` property set to `true` causes the Fusion web application to render partial content. For information about page rendering, see [Rerendering Partial Page Content](#).
- A component that renders a toggle icon for end users to display or hide content hosts the `popup` component. Examples include the `showDetailItem` and `panelTabbed` components. For information about the use of components that render toggle icons, see [Displaying and Hiding Contents Dynamically](#).
- Failover occurs when the Fusion web application displays an inline popup. During failover, the Fusion web application replaces the entire page.

You can change the default behavior described in the previous list by disabling the automatic cancellation of an inline popup component. This means that the Fusion web application does not automatically cancel the inline popup if any of the above events occur. Instead, the Fusion web applications restores the inline popup.

How to Disable the Automatic Cancellation of an Inline Popup

You disable the automatic cancellation of an inline popup by setting the `popup` component's `autoCancel` property to `disabled`.

Before you begin:

It may be helpful to understand how other components can affect functionality. See [Controlling the Automatic Cancellation of Inline Popups](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To control the automatic cancellation of inline popups:

1. In the Structure window, right-click the `af:popup` component for which you want to configure the automatic cancellation behavior and choose **Go to Properties**.
2. In the Properties window, expand the **Common** section and select **disabled** from the **AutoCancel** dropdown list.

What Happens When You Disable the Automatic Cancellation of an Inline Popup

JDeveloper sets the `af:popup` component `autoCancel` property's value to `disabled`, as shown in the following example. At runtime, the Fusion web application restores an inline popup after it rerenders a page if the inline popup displayed before invocation of the command to rerender the page.

```
<af:popup id="p1" autoCancel="disabled">  
  ...  
</af:popup>
```

Resetting Input Fields in a Popup

The ADF Faces `resetListener` component is a declarative way to reset input component values triggered from any server-side event. Using this component, you can allow end users to reset input values in an input field.

You can use the `resetListener` component with a `popup` component to allow end users to reset input values in an input field. Example use cases where you may want to implement this functionality for input components that render in a `popup` component include:

- Permitting end users to reset an incorrect value that they previously entered
- Removing values where the `popup` component invokes a `popupCanceledEvent` before the application submits the values to the server that an end user entered.

End user gestures that invoke a `popupCancelEvent` include clicking a button (for example, a button labelled **Close**), the cancel icon in the title bar of a popup dialog or pressing the Esc key.

Depending on how you configure the `popup` component, data may be cached on the client. For example, if you set the `popup` component's `contentDelivery` attribute to `immediate`, the application always caches data on the client.

For information about how the setting that you choose for the `contentDelivery` attribute determines the content delivery strategy for your `popup` component, see [Declaratively Creating Popups](#) and [What Happens at Runtime: Popup Component Events](#).

[Declaratively Invoking a Popup](#) shows the metadata for a popup component where the `contentDelivery` attribute is set to `immediate` and the user's popup renders a `dialog` component with preconfigured controls that raise `dialogEvents`, as described in [How to Create a Dialog](#). In this scenario, data that the end user entered is cached on the client. The application does not submit data that you want to reset to the server. Also, the preconfigured controls rendered by the `dialog` component may prevent the popup from closing if they encounter validation errors.

For information about using the `resetListener` component independently of a `popup` component, see [How to Use an Action Component to Reset Input Fields](#).

Note:

Setting the `resetListener` component's `type` attribute to `popupCanceled` provides the same functionality as setting the `popup` component's `resetEditableValues` attribute to `whenCanceled`. For information about setting the `resetEditableValues` attribute of the `popup` component, see [How to Create a Dialog](#).

Example 16-3 The `resetListener` Tag on the `Popup` Component

```
<af:popup id="popup" contentDelivery="immediate">
  <af:resetListener type="popupCanceled"/>
</af:popup>
```

How to Reset the Input Fields in a Popup

You enable end users to reset the data in a popup's input fields to `null` by setting the `resetListener` component's `type` attribute to `popupCanceled`.

Before you begin:

It may be helpful to understand the use cases for which you can configure this functionality in a `popup` component. See [Resetting Input Fields in a Popup](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Popup Dialogs, Menus, and Windows](#).

To reset the input fields in a popup:

1. Create the type of popup dialog that you require, as described in [Declaratively Creating Popups](#).
2. In the Components window, from the Operations panel, in the Listeners group, drag and drop a **Reset Listener** as a direct child to the `popup` component.
3. In the Insert Reset Listener dialog that JDeveloper displays, enter `popupCanceled` as the type of event that the `resetListener` component responds to.

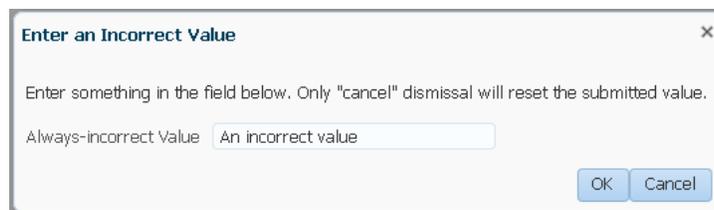
Click **Help** in the Insert Reset Listener dialog to view a complete list of supported values.

What Happens When You Configure a Popup to Reset Its Input Fields

JDeveloper writes entries similar to those shown in [Example 16-4](#) when you configure a `popup` component and a `resetListener` component to allow end users to reset the input field(s) in the `popup` component to `null`.

At runtime, an end user gesture that raises a `popupCanceled` event results in the `resetListener` component resetting values in the input fields of the `popup` component to `null`, as illustrated in [Figure 16-11](#).

Figure 16-11 Popup Component Resetting Input Fields



Example 16-4 Popup Component Configured to Reset Input Fields Using Reset Listener

```
<af:popup id="popupDialog" contentDelivery="lazyUncached"
  popupCanceledListener="#{demoInput.resetPopupClosed}">
  <af:dialog title="Enter an Incorrect Value">
    <af:inputText id="it2" label="Always-incorrect Value" value="#{demoInput.value}">
      <f:validator binding="#{demoInput.obstinateValidator2}" />
    </af:inputText>
```

```
</af:dialog>  
  <af:resetListener type="popupCanceled" />  
</af:popup>
```

17

Using a Calendar Component

This chapter describes how to use the ADF Faces calendar component to create a calendar application.

This chapter includes the following sections:

- [About Creating a Calendar Component](#)
- [Creating the Calendar](#)
- [Configuring the Calendar Component](#)
- [Adding Functionality Using Popup Components](#)
- [Customizing the Toolbar](#)
- [Styling the Calendar](#)

About Creating a Calendar Component

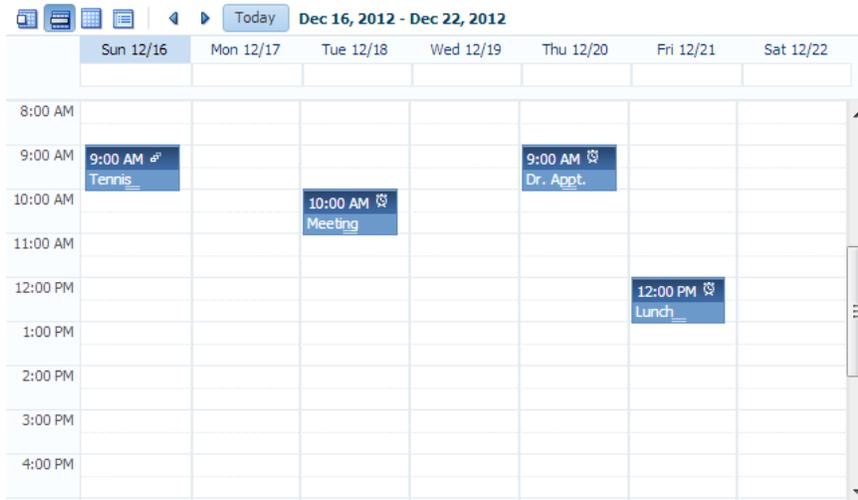
ADF Faces provides a calendar component that displays user activities. You can add functionality to this component so that the user can edit, create, and delete activities from the calendar.

ADF Faces includes a calendar component that by default displays activities in daily, weekly, monthly, or list views for a given provider or providers (a provider is the owner of an activity). [Figure 17-1](#) shows an ADF Faces calendar in weekly view mode with some sample activities.

 **Note:**

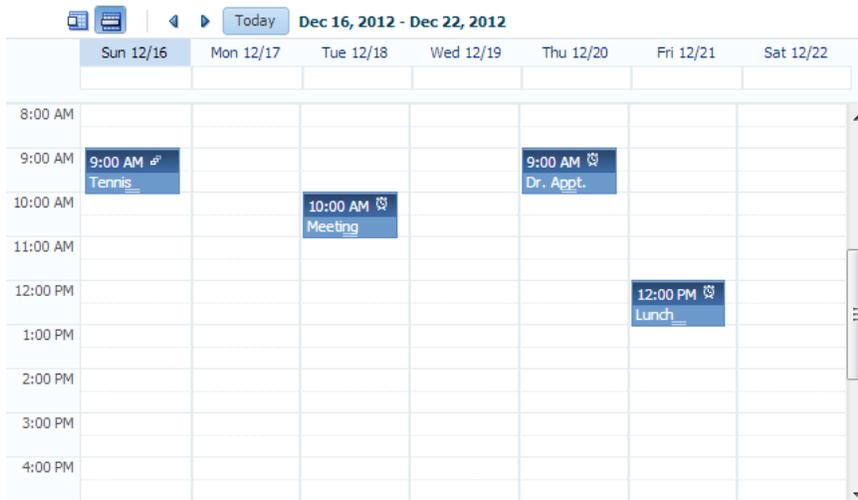
Printing is possible in the printable page mode for all types of views and the scrollbars will be dropped in the printed file.

Figure 17-1 ADF Faces Calendar Showing Weekly View



You can configure the calendar so that it displays only a subset of views. For example, you may not want your calendar to use the month and list views. You can configure it so that only the day and week views are available, as shown in [Figure 17-2](#). Because only day and week views are available, those are the only buttons displayed in the toolbar.

Figure 17-2 Calendar Configured to Use Only Week and Day Views



By default, the calendar displays dates and times based on the locale set in the `trinidad-config.xml` file using the `formatting-locale` parameter. See [Configuration in trinidad-config.xml](#). If a locale is not specified in that file, then it is based on the locale sent by the browser. For example, in the United States, by default, the start day of the week is Sunday, and 2 p.m. is shown as 2:00 PM. In France, the default start day is Monday, and 2 p.m. is shown as 14:00. The time zone for the calendar is also based on the `time-zone` parameter setting in `trinidad-config.xml`. You can override

the default when you configure the calendar. See [Configuring the Calendar Component](#).

The calendar includes a toolbar with built-in functionality that enables a user to change the view (between daily, weekly, monthly, or list), go to the previous or next day, week, or month, and return to today. The toolbar is fully customizable. You can choose which buttons and text to display, and you can also add buttons or other components. See [Customizing the Toolbar](#).

 **Tip:**

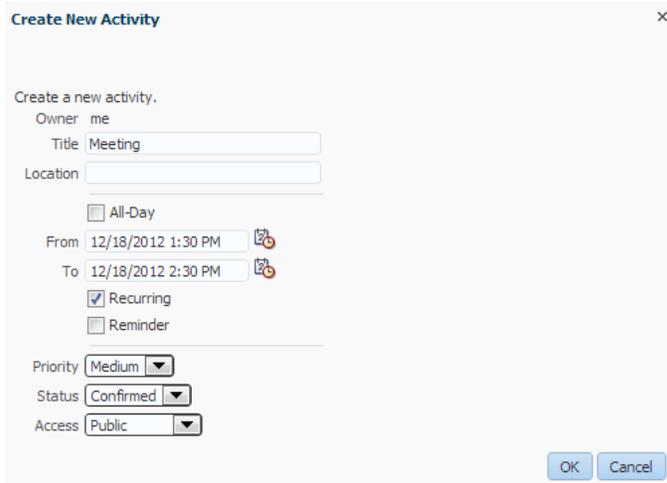
When these toolbar buttons are used, attribute values on the calendar are changed. You can configure these values to be persisted so that they remain for the user during the duration of the session. See [Allowing User Customization on JSF Pages](#).

You can also configure your application so that the values will be persisted and used each time the user logs into the system. For this persistence to take place, your application must use the Fusion technology stack. See [Allowing User Customizations at Runtime in *Developing Fusion Web Applications with Oracle Application Development Framework*](#).

The calendar component displays activities based on the activities and the provider returned by the `CalendarModel` class. By default, the calendar component is read-only. That is, it can display only those activities that are returned. You can add functionality within supported facets of the calendar so that users can edit, create, and delete activities. When certain events are invoked, popup components placed in these corresponding facets are opened, which enable the user to act on activities or the calendar.

For example, when a user clicks on an activity in the calendar, the `CalendarActivityEvent` is invoked and the popup component in the `ActivityDetail` facet is opened. You might use a dialog component that contains a form where users can view and edit the activity, as shown in [Figure 17-3](#).

Figure 17-3 Dialog Implemented to Edit an Activity



Create a new activity.

Owner me

Title Meeting

Location

All-Day

From 12/18/2012 1:30 PM

To 12/18/2012 2:30 PM

Recurring

Reminder

Priority Medium

Status Confirmed

Access Public

OK Cancel

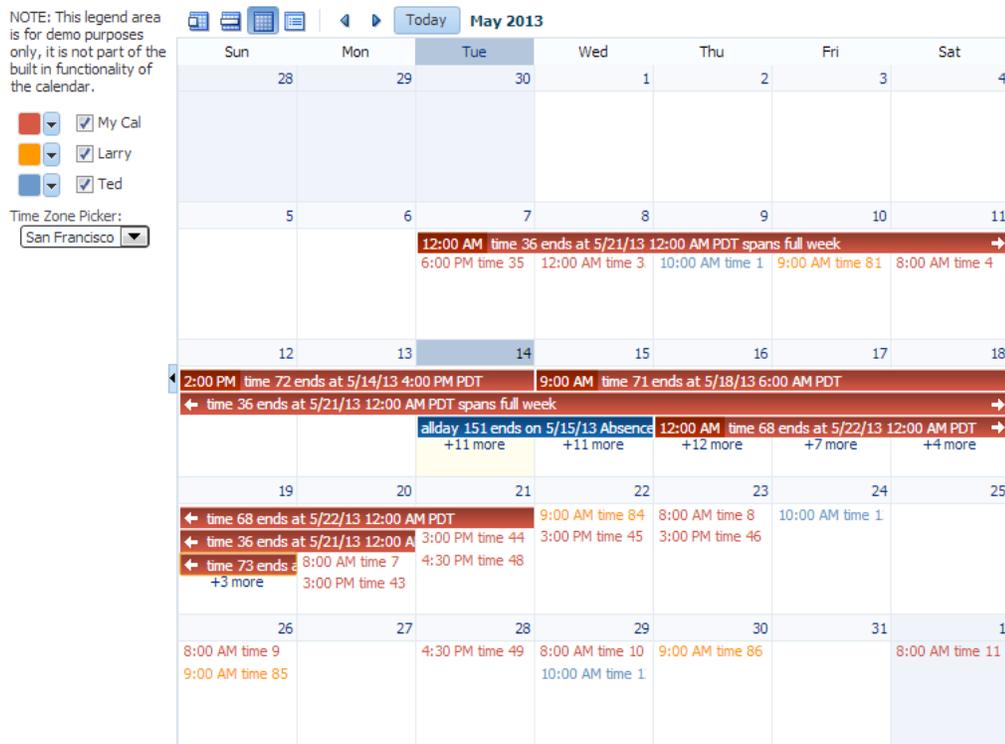
For information about implementing additional functionality using events, facets, and popup components, see [Adding Functionality Using Popup Components](#).

The calendar component supports the ADF Faces drag and drop architectural feature. Users can drag activities to different areas of the calendar, executing either a copy or a move operation, and can also drag handles on the activity to change the duration of the activity. For information about adding drag and drop functionality, see [Adding Drag and Drop Functionality to a Calendar](#).

By default, the calendar displays activities using a blue color ramp. A color ramp is a set of colors in a color family and is used to represent the different states of activities. In the default calendar, for a short-duration activity shown in the daily view, the time of an activity is shown with a dark blue background, while the title of the activity is shown with a light blue background, as shown in [Figure 17-1](#). You can customize how the activities are displayed by changing the color ramp.

Each activity is associated with a provider, that is, an owner. If you implement your calendar so that it can display activities from more than one provider, you can also style those activities so that each provider's activity shows in a different color, as shown in [Figure 17-4](#).

Figure 17-4 Month View with Activities from Different Providers



Calendar Use Cases and Examples

The calendar component provides the features you need to implement calendar-related functions such as creating activities in daily, weekly, monthly or list view. It features a customizable toolbar that can be used for switching views. It has configurable start of the week and start of the day functions. Like other ADF Faces components, it supports skinning in order to customize its style and appearance.

You can create popups by inserting them into the calendar facets to add more functionality. You can also implement the calendar so the user can drag and drop activities from one area to another within the calendar.

The calendar uses the `CalendarModel` class to display the activities for a given time period. You must create your own implementation of the model class for your calendar. If your application uses the Fusion technology stack, you can create ADF Business Components over your data source that represents the activities, and the model will be created for you. You can then declaratively create the calendar, and it will automatically be bound to that model. See *Using the ADF Faces Calendar Component* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

If your application does not use the Fusion technology stack, then you create your own implementation of the `CalendarModel` class and the associated `CalendarActivity` and `CalendarProvider` classes. The classes are abstract classes with abstract methods. You must provide the functionality behind the methods, suitable for your implementation of the calendar. See [Creating the Calendar](#).

Additional Functionality for the Calendar

You may find it helpful to understand other ADF Faces features before you implement your calendar component. Additionally, once you have added a calendar component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that calendar components can use.

- **Client components:** Components can be client components. To work with the components on the client, see [Using ADF Faces Client-Side Architecture](#).
- **JavaScript APIs:** All components have JavaScript client APIs that you can use to set or get property values. See the *JavaScript API Reference for Oracle ADF Faces*.
- **Events:** Components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- You can display tips and messages, as well as associate online help with a calendar component. See [Displaying Tips, Messages, and Help](#).
- You may want other components on a page to update based on selections you make from a calendar component. See [Using the Optimized Lifecycle](#).
- You can change the appearance using skins. See [Customizing the Appearance Using Styles and Skins](#).
- You can make your components accessible. See [Developing Accessible ADF Faces Pages](#).
- Instead of entering values for attributes that take strings as values, you can use property files. These files enable you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- You can create popups for additional functionality. For information about using these events to provide additional functionality, see [Adding Functionality Using Popup Components](#).
- If your application uses ADF Model, then you can create automatically bound forms using data controls (whether based on ADF Business Components or other

business services). See *Creating a Basic Databound Page in Developing Fusion Web Applications with Oracle Application Development Framework*.

Creating the Calendar

The ADF Faces calendar component allows a user to view their activities by day, week, month, or by list view. To use a calendar component, you must define the logic required by the calendar in Java classes.

Before you can add a calendar component to a page, you must implement the logic required by the calendar in Java classes that extend ADF Faces calendar abstract classes. After you create the classes, you can add the calendar to a page.

Note:

If your application uses the Fusion technology stack, implement the calendar classes using ADF Business Components. This will enable you to declaratively create and bind your calendar component. See *Using the ADF Faces Calendar Component in Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you implement your logic, it helps to have an understanding of the `CalendarModel` and `CalendarActivity` classes, as described in the following section.

Calendar Classes

The calendar component must be bound to an implementation of the `CalendarModel` class. The `CalendarModel` class contains the data for the calendar. This class is responsible for returning a collection of calendar activities, given the following set of parameters:

- **Provider ID:** The owner of the activities. For example, you may implement the `CalendarModel` class such that the calendar can return just the activities associated with the owner currently in session, or it can also return other owners' activities.
- **Time range:** The expanse of time for which all activities that begin within that time should be returned. A date range for a calendar is inclusive for the start time and exclusive for the end time (also known as half-open), meaning that it will return all activities that intersect that range, including those that start before the start time, but end after the start time (and before the end time).

A calendar activity represents an object on the calendar, and usually spans a certain period of time. The `CalendarActivity` class is an abstract class whose methods you can implement to return information about the specific activities.

Activities can be recurring, have associated reminders, and be of a specific time type (for example, `TIME` (with a start and end time) or `ALLDAY`). Activities can also have start and end dates, a location, a title, and a tag.

The `CalendarProvider` class represents the owner of an activity. A provider can be either enabled or disabled for a calendar.

How to Create a Calendar

Create your own implementations of the `CalendarModel` and `CalendarActivity` classes and implement the abstract methods to provide the logic.

Before you begin:

It may be helpful to have an understanding of the `CalendarModel` and `CalendarActivity` classes. See [Calendar Classes](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To create the calendar model classes:

1. Create a managed bean that will return an instance of the `oracle.adf.view.rich.model.CalendarModel` class. This instance must:
 - Extend the `oracle.adf.view.rich.model.CalendarModel` class.
 - Implement the abstract methods.
For information about the `CalendarModel` class, see the *Java API Reference for Oracle ADF Faces*.
 - Implement any other needed functionality for the calendar. For example, you might add logic that sets the time zone, as in the `oracle.adfdemo.view.calendar.rich.model.DemoCalendarBean` managed bean in the ADF Faces Components Demo application (for information about downloading and installing the demo application, see [ADF Faces Components Demo Application](#)).

For information about creating managed beans, see [Creating and Using Managed Beans](#).

2. Create a managed bean that will return an instance of the `oracle.adf.view.rich.model.CalendarActivity` class. This instance must:
 - Extend the `oracle.adf.view.rich.model.CalendarActivity` class.
 - Implement the abstract methods.
For information about the `CalendarActivity` class, see the *Java API Reference for Oracle ADF Faces*.
 - Implement any other required functionality for the calendar activities. For an example, see the `oracle.adfdemo.view.calendar.rich.model.DemoCalendarActivity` managed bean in the ADF Faces Components Demo application.

Tip:

If you want to style individual instances of an activity (for example, if you want each provider's activities to be displayed in a different color), then use the `getTags` method to return a tag that represents what group the activity belongs to (for example, using the provider ID). See [How to Style Activities](#).

3. Create a managed bean that will return an instance of the `oracle.adf.view.rich.model.CalendarProvider` class. This instance must:
 - Extend the `oracle.adf.view.rich.model.CalendarProvider` class.
 - Implement the abstract methods.
For information about the `CalendarProvider` class, see the *Java API Reference for Oracle ADF Faces*.
 - Implement any other required functionality for the calendar providers.

To create the calendar component:

1. In the Components window, from the Data Views panel, drag a **Calendar** and drop it onto the JSF page.

 **Tip:**

The `calendar` component can be stretched by any parent component that can stretch its children. If the calendar is a child component to a component that cannot be stretched, it will use a default width and height, which cannot be stretched by the user at runtime. However, you can override the default width and height using inline style attributes. For information about the default height and width, see [Configuring the Calendar Component](#). For information about stretching components, see [Geometry Management and Component Stretching](#).

2. Expand the Calendar Data panel of the Properties window, and enter an EL expression for **Value** that resolves to the managed bean that extends the `CalendarModel` class.

Configuring the Calendar Component

The ADF Faces calendar component can be configured using the Properties window. You can configure the component using the available views.

Configure the many display attributes for the calendar, for example, the time displayed at the beginning of a day.

How to Configure the Calendar Component

You configure the calendar using the Properties window.

Before you begin:

It may be helpful to have an understanding of the calendar component. See [Configuring the Calendar Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To configure a calendar:

1. In the Properties window, view the attributes for the calendar. Use the Help button to display the complete tag documentation for the `calendar` component.

2. With the `calendar` component selected, expand the Common section of the Properties window, and set the following:

- **AvailableViews:** Select the available views. The value can be one of or a combination of the following:

- `all`
- `day`
- `week`
- `month`
- `list`

If you want to enter more than one value, enter the values with a space between. For example, if you want the calendar to use day and week views, you would enter the following:

```
day week
```

 **Note:**

If `all` is entered, then all views are available, regardless if one is left out of the list.

The corresponding buttons will automatically be displayed in the toolbar, in the order they appear listed for the `availableViews` attribute.

 **Note:**

In order to handle an overflow of tasks for a given day in the `month` view, if you enter `month` and do not also enter `all`, then you must also enter `day`.

- **View:** Select the view that should be the default when the calendar is displayed. Users change this value when they click the corresponding button in the calendar's toolbar. Valid values are:

- `day`
- `list`
- `month`
- `week`

- **StartDayOfWeek:** Enter the day of the week that should be shown as the starting day, at the very left in the monthly or weekly view. When not set, the default is based on the user's locale. Valid values are:

- `sun`
- `mon`
- `tue`
- `wed`

- thu
- fri
- sat
- **StartHour:** Enter a number that represents the hour (in 24-hour format, with 0 being midnight) that should be displayed at the top of the `day` and `week` view. While the calendar renders all 24 hours of the day, the calendar will scroll to the `startHour`, displaying this hour at the top of the view. The user can scroll above that time to view activities that start before the `startHour` value.
- **ListType:** Select how you want the `list` view to display activities. Valid values are:
 - `day`: Shows activities only for the active day.
 - `dayCount`: Shows a number of days including the active day and after, based on the value of the `listCount` attribute.
 - `month`: Shows all the activities for the month to which the active day belongs.
 - `week`: Shows all the activities for the week to which the active day belongs.
- **ListCount:** Enter the number of days' activities to display (used only when the `listType` attribute is set to `dayCount`).

Figure 17-5 shows a calendar in list view with the `listType` set to `dayCount` and the `listCount` value set to 14.

Figure 17-5 List View Using dayCount Type

Day	Date	Time	Duration	Frequency	Color	Owner
Tuesday	Apr 23	4:30 PM	time 50	recurring weekly 3	Red	My Cal
Wednesday	Apr 24	8:00 AM	time 10		Red	My Cal
		10:00 AM	time 124		Blue	Ted
Thursday	Apr 25	9:00 AM	time 87		Yellow	Larry
Saturday	Apr 27	8:00 AM	time 11		Red	My Cal
Monday	Apr 29	9:00 AM	time 88		Yellow	Larry
		10:00 AM	time 125		Blue	Ted
Tuesday	Apr 30	8:00 AM	time 12		Red	My Cal
		4:30 PM	time 51	recurring weekly 4	Red	My Cal
Friday	May 3	8:00 AM	time 13		Red	My Cal
		9:00 AM	time 89		Yellow	Larry
Saturday	May 4	10:00 AM	time 126		Blue	Ted
Monday	May 6	8:00 AM	time 14		Red	My Cal

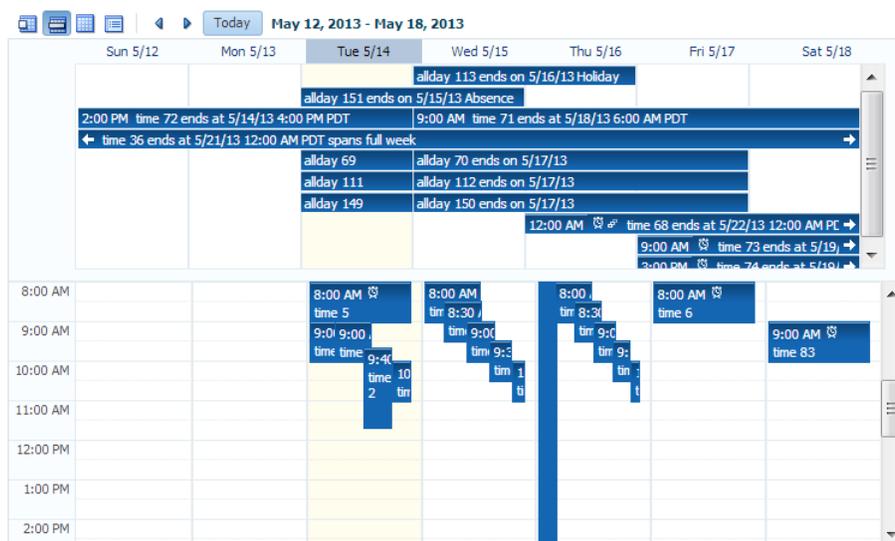
3. Expand the Calendar Data section of the Properties window, and set the following:
 - **ActiveDay:** Set the day used to determine the date range that is displayed in the calendar. By default, the active day is today's date for the user. Do not change this if you want today's date to be the default active day when the calendar is first opened.

Note that when the user selects another day, this becomes the value for the `activeDay` attribute. For example, when the user first accesses the calendar, the current date is the active day. The user can select another day to be the active day by clicking on the day link in the month view. The active day also changes when the user selects a different month or year.

- **TimeZone:** Set the time zone for the calendar. If not set, the value is taken from `AdfFacesContext`. The valid value is a `java.util.TimeZone` object. By default, time is displayed based on the `formatting-locale` parameter in the `trinidad-config.xml` file. See [Configuration in trinidad-config.xml](#).
4. Expand the Appearance section of the Properties window and set the following:
- **AllDayActivityOrder:** Control the display of all-day activities by specifying a list of strings that correspond to tags on the activities. Activities will be grouped by tags and display in the order specified by `allDayActivityOrder`. Activities without any tags, or whose tags are not listed in `allDayActivityOrder`, will display together in a default group. Within each group, the calendar displays rows with the most activities first. For example, if `allDayActivityOrder = holiday absence`, then all-day activities will be displayed in this order:
 - Activities with the holiday tag
 - Activities with the absence tag
 - Activities with no tags

Figure 17-6 shows a calendar with `allDayActivityOrder` set to `holiday absence`. Activities tagged holiday appear at the top, followed by activities tagged absence. The remaining activities are then displayed; rows with the most activities appear first.

Figure 17-6 Calendar Display Using allDayActivityOrder



- **HourZoom:** Set the zoom factor for time cells to be displayed in the calendar. The zoom factor applies to the height of the hour in `day` or `week` view. Valid values are `auto` or a non-zero positive number (including fractions). By default, the value is 1.

A value greater than 1 will scale up the calendar by the specified factor. For example, a value of 2 will scale up the calendar by 200%. A value of 0.5 will scale down the calendar by 50%. When set to `auto` the calendar will scale by an optimal factor for best viewing, ensuring that tightly scheduled non-overlapping activities will not display overlapping each other for lack of vertical space.

- **TimeSlotsPerHour:** Set the number of time slots to display per hour in day or week view. Time slots are minor divisions per hour, indicated by a dotted line splitting the hour into shorter intervals. For example, the value 4 will render four time slots per hour, measuring 15 minutes each. Valid values are `auto` or a non-zero positive whole number. By default, the value is `auto`.

When set to `auto` the calendar will use the skin property `-tr-time-slots-per-hour`. For example, `af|calendar {-tr-time-slots-per-hour: 4}` will render a minor division (dotted line) at 15-minute intervals.

5. If you want the user to be able to drag and resize the calendar regions, expand the Other section of the Properties window and set the `splitterCollapsed` and `splitterPosition` attributes. The splitter separates the all-day and timed activities areas in the day and week views of the calendar (it has no effect in month and list views). By default `splitterCollapsed` is `false`, which means that both the all-day and timed activities areas are displayed. When the splitter is collapsed (`splitterCollapsed = true`), the all-day activities area is hidden and the timed activities area stretches to fill all available vertical space. The `splitterPosition` attribute specifies the initial height in pixels of the all-day activities area; the timed activities area gets the remaining space. Valid values are `auto` or a non-zero positive whole number. By default, the value is `auto`. For information, see the tag documentation for the `calendar` component.
6. If you want the user to be able to drag a handle on an existing activity to expand or collapse the time period of the activity, then implement a handler for `CalendarActivityDurationChangeListener`. This handler should include functionality that changes the end time of the activity. If you want the user to be able to move the activity (and, therefore, change the start time as well as the end time), then implement drag and drop functionality. See [Adding Drag and Drop Functionality to a Calendar](#).

You can now add the following functionality:

- Create, edit, and delete activities using popup components. See [Adding Functionality Using Popup Components](#).
- Move activities around on the calendar. See [Adding Drag and Drop Functionality to a Calendar](#).
- Change or add to the toolbar buttons in the toolbar. See [Customizing the Toolbar](#).
- Change the appearance of the calendar and events. See [Styling the Calendar](#).

What Happens at Runtime: Calendar Events and PPR

The calendar has two events that are used in conjunction with facets to provide a way to easily implement additional functionality needed in a calendar, such as editing or adding activities. These two events are `CalendarActivityEvent` (invoked when an action occurs on an activity) and `CalendarEvent` (invoked when an action occurs on the calendar itself). For information about using these events to provide additional functionality, see [Adding Functionality Using Popup Components](#).

The calendar also supports events that are fired when certain changes occur. The `CalendarActivityDurationChangeEvent` is fired when the user changes the duration of an activity by making changes to the start or end time. The `CalendarDisplayChangeEvent` is fired when the value of a display attribute changes. For example, if a user changes the `view` attribute from `day` to `month`, the calendar is

rendered automatically because the calendar component becomes a partial page rendering (PPR) target, triggering an immediate refresh.

Adding Functionality Using Popup Components

You can add ADF Faces popup components to a calendar component. To add functionality, you must create the popups and associated components in the associated facets.

When a user acts upon an activity, a `CalendarActivityEvent` is fired. This event causes the popup component contained in a facet to be displayed, based on the user's action. For example, if the user right-clicks an activity, the `CalendarActivityEvent` causes the popup component in the `activityContextMenu` to be displayed. The event is also delivered to the server, where a configured listener can act upon the event. You create the popup components for the facets (or if you do not want to use a popup component, implement the server-side listener). It is in these popup components and facets where you can implement functionality that will enable users to create, delete, and edit activities, as well as to configure their instances of the calendar.

[Table 17-1](#) shows the different user actions that invoke events, the event that is invoked, and the associated facet that will display its contents when the event is invoked. The table also shows the component you must use within the popup component. You create the popup and the associated component within the facet, along with any functionality implemented in the handler for the associated listener. If you do not insert a popup component into any of the facets in the table, then the associated event will be delivered to the server, where you can act on it accordingly by implementing handlers for the events.

Table 17-1 Calendar Faces Events and Associated Facets

User Action	Event	Associated Facet	Component to Use in Popup
Right-click an activity.	<code>CalendarActivityEvent</code>	<code>activityContextMenu</code> : The enclosed popup component can be used to display a context menu, where a user can choose some action to execute against the activity (for example, edit or delete).	<code>menu</code>
Select an activity and press the Delete key.	<code>CalendarActivityEvent</code>	<code>activityDelete</code> : The enclosed popup component can be used to display a dialog that allows the user to delete the selected activity.	<code>dialog</code>
Click an activity or select an activity and press the Enter key.	<code>CalendarActivityEvent</code>	<code>activityDetail</code> : The enclosed popup component can be used to display the activity's details.	<code>dialog</code>
Hover over an activity.	<code>CalendarActivityEvent</code>	<code>activityHover</code> : The enclosed popup component can be used to display high-level information about the activity.	<code>noteWindow</code>

Table 17-1 (Cont.) Calendar Faces Events and Associated Facets

User Action	Event	Associated Facet	Component to Use in Popup
Right-click the calendar (not an activity or the toolbar).	CalendarEvent	contextMenu: The enclosed popup component can be used to display a context menu for the calendar.	menu
Click or double-click any free space in the calendar (not an activity).	CalendarEvent	create: The enclosed popup component can be used to display a dialog that allows a user to create an activity.	dialog

How to Add Functionality Using Popup Components

To add functionality, create the popups and associated components in the associated facets.

Before you begin:

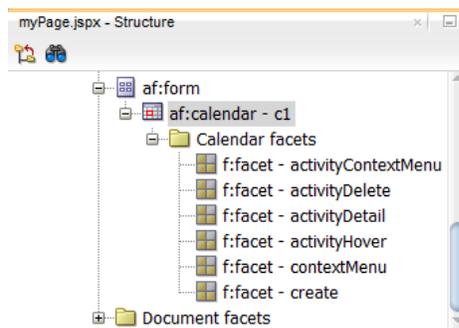
It may be helpful to have an understanding of popup components. See [Adding Functionality Using Popup Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To add functionality using popup components:

1. In the Structure window, expand the **af:calendar** component node so that the calendar facets are displayed, as shown in [Figure 17-7](#).

Figure 17-7 Calendar Facets in the Structure Window



2. Based on [Table 17-1](#), create popup components in the facets that correspond to the user actions for which you want to provide functionality. For example, if you want users to be able to delete an activity by clicking it and pressing the Delete key, you add a popup dialog to the `activityDelete` facet.

To add a popup component, right-click the facet in the Structure window and choose **Insert inside facetName > Popup**.

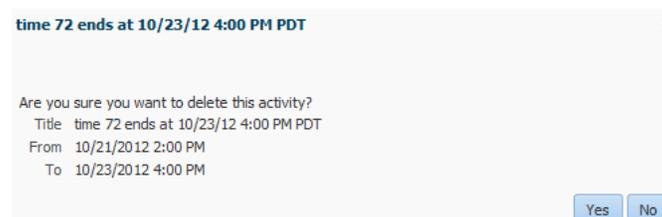
For information about creating popup components, see [Using Popup Dialogs, Menus, and Windows](#).

The following example shows the JSF code for a dialog popup component used in the `activityDelete` facet.

```
<f:facet name="activityDelete">
  <af:popup id="delete" contentDelivery="lazyUncached">
    <!-- don't render if the activity is null -->
    <af:dialog dialogListener="#{calendarBean.deleteListener}"
      affirmativeTextAndAccessKey="Yes" cancelTextAndAccessKey="No"
      rendered="#{calendarBean.currActivity != null}">
      <af:spacer height="20"/>
      <af:outputText value="Are you sure you want to delete this activity?"/>
      <af:panelFormLayout>
        <af:inputText label="Title" value="#{calendarBean.currActivity.title}"
          readOnly="true"/>
        <af:inputDate label="From" value="#{calendarBean.currActivity.from}"
          readOnly="true">
          <af:convertDateTime type="date" dateStyle="short"
            timeZone="#{calendarBean.timeZone}"
            pattern="#{calendarBean.currActivity.dateTimeFormat}"/>
        </af:inputDate>
        <af:inputDate label="To" value="#{calendarBean.currActivity.to}"
          readOnly="true">
          <af:convertDateTime type="date" dateStyle="short"
            timeZone="#{calendarBean.timeZone}"
            pattern="#{calendarBean.currActivity.dateTimeFormat}"/>
        </af:inputDate>
        <af:inputText label="Location" readOnly="true"
          rendered="#{calendarBean.currActivity.location != null}"
          value="#{calendarBean.currActivity.location}"/>
      </af:panelFormLayout>
    </af:dialog>
  </af:popup>
</f:facet>
```

Figure 17-8 shows how the dialog is displayed when a user clicks an activity and presses the Delete key.

Figure 17-8 Delete Activity Dialog



3. Implement any needed logic for the `calendarActivityListener`. For example, if you are implementing a dialog for the `activityDeleteFacet`, then implement logic in the `calendarActivityListener` that can save-off the current activity so that when you implement the logic in the dialog listener (in the next step), you will know which activity to delete. The following example shows the `calendarActivityListener` for the `calendar.jspx` page in the ADF Faces Components Demo application.

```

public void activityListener(CalendarActivityEvent ae)
{
    CalendarActivity activity = ae.getCalendarActivity();
    if (activity == null)
    {
        // no activity with that id is found in the model
        setCurrActivity(null);
        return;
    }

    setCurrActivity(new DemoCalendarActivityBean((DemoCalendarActivity)activity,
        getTimeZone()))

```

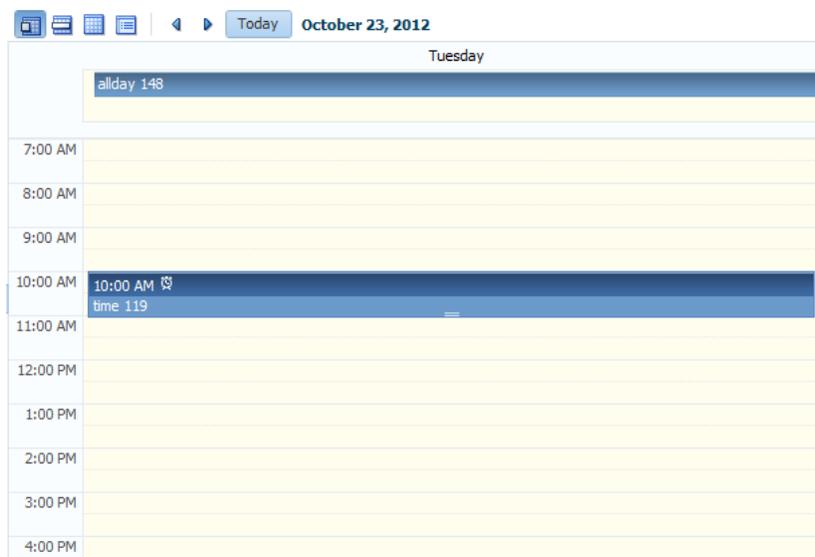
4. Implement the logic for the popup component in the handler for the popup event. For example, for the delete dialog, implement a handler for the `dialogListener` that actually deletes the activity when the dialog is dismissed. For information about creating dialogs and other popup components, see [Using Popup Dialogs, Menus, and Windows](#).

Customizing the Toolbar

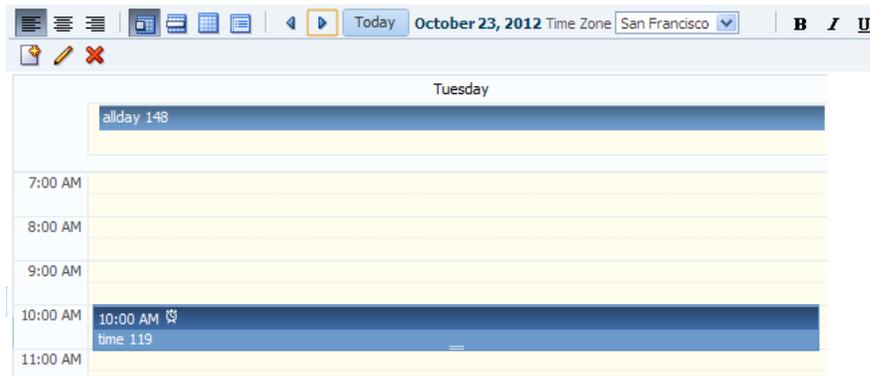
You can customize the ADF Faces calendar component toolbar to change the display date, to display additional toolbar buttons, or to add a text describing the current view. To customize the toolbar, you must place the toolbar and toolbar buttons in the custom facets that you create.

By default, the toolbar in the calendar enables the user to change the view between day, week, month, and list, go to the next or previous item in the view, or go to the present day. The toolbar also displays a text description of the current view. For example in the day view, it displays the active date, as shown in [Figure 17-9](#).

Figure 17-9 Toolbar in Day View of a Calendar



[Figure 17-10](#) shows a toolbar that has been customized. It has added toolbar buttons, including buttons that are right-aligned on the top toolbar, and buttons in a second toolbar.

Figure 17-10 Customized Toolbar for a Calendar

How to Customize the Toolbar

Place the toolbar and toolbar buttons you want to add in custom facets that you create. Then, reference the facet (or facets) from an attribute on the calendar, along with keywords that determine how or where the contained items should be displayed.

Before you begin:

It may be helpful to have an understanding of calendar toolbar customization. See [Customizing the Toolbar](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To customize the toolbar:

1. In the JSF page of the Components window, from the Layout (Core Structure) panel, drag and drop a **Facet** for each section of the toolbar you want to add. For example, to add the custom buttons shown in [Figure 17-10](#), you would add four facet tags. Ensure that each facet has a unique name for the page.

Tip:

To ensure that there will be no conflicts with future releases of ADF Faces, start all your facet names with `customToolbar`. For example, the section of the toolbar that contains the alignment buttons shown in [Figure 17-10](#) are in the `customToolbarAlign` facet.

2. In the ADF Faces page of the Components window, from the Menus and Toolbars panel, drag and drop a **Toolbar** to each facet and add toolbar buttons and configure the buttons and toolbar as needed. For information about toolbars and toolbar buttons, see [Using Toolbars](#).
3. Select the calendar, and in the Properties window, from the dropdown menu next to the `ToolboxLayout` attribute, choose **Edit**.
4. In the Edit Property: `ToolboxLayout` dialog set the value for this attribute. It should be a list of the custom facet names, in the order in which you want the contents in

the custom facets to appear. In addition to those facets, you can also include all, or portions of the default toolbar, using the following keywords:

- `all`: Displays all the toolbar buttons and text in the default toolbar
- `dates`: Displays only the previous, next, and today buttons
- `range`: Displays only the string showing the current date range
- `views`: Displays only the buttons that allows the user to change the view

 **Note:**

If you use the `all` keyword, then the `dates`, `range`, and `views` keywords are ignored.

For example, if you created two facets named `customToolbar1` and `customToolbar2`, and you wanted the complete default toolbar to appear in between your custom toolbars, the value of the `toolbarLayout` attribute would be the following list items:

- `customToolbar1`
- `all`
- `customToolbar2`

You can also determine the layout of the toolbars using the following keywords:

- `newline`: Places the toolbar in the next named facet (or the next keyword from the list in the `toolbarLayout` attribute) on a new line. For example, if you wanted the toolbar in the `customToolbar2` facet to appear on a new line, the list would be:

```
- customToolbar1
- all
- newline
- customToolbar2
```

If instead, you did not want to use all of the default toolbar, but only the views and dates sections, and you wanted those to each appear on a new line, the list would be:

```
- customToolbar1
- customToolbar2
- newline
- views
- newline
- dates
```

- `stretch`: Adds a spacer component that stretches to fill up all available space so that the next named facet (or next keyword from the default toolbar) is displayed as right-aligned in the toolbar. The following example shows the value of the `toolbarLayout` attribute for the toolbar displayed in [Figure 17-10](#),

along with the toolbar placed in the `customToolbarAlign` facet. Note that the toolbar buttons displayed in the `customToolbarBold` facet are right-aligned in the toolbar because the keyword `stretch` is named before the facet.

```
<af:calendar binding="#{editor.component}" id="calendar1"
  value="#{calendarBean.calendarModel}"
  timeZone="#{calendarBean.timeZone}"
  toolboxLayout="customToolbarAlign all customToolbarTZ stretch
                customToolbarBold newline customToolbarCreate"
. . .
<f:facet name="customToolbarAlign">
  <af:toolbar>
    <af:button id="alignLeft" shortDesc="align left"
      icon="/images/alignleft16.png" type="radio"
      selected="true"/>
    <af:button id="alignCenter" shortDesc="align center"
      icon="/images/aligncenter16.png" type="radio"
      selected="false"/>
    <af:button id="alignRight" shortDesc="align right"
      icon="/images/alignright16.png" type="radio"
      selected="false"/>
  </af:toolbar>
</f:facet>
. . .
</af:calendar>
```

Styling the Calendar

The ADF Faces calendar component allows you to apply available styles and select skins to change its appearance. You can also add `activityStyles` and `dateCustomizer` attributes to enable users to ease the styling instances of a calendar.

Like other ADF Faces components, the calendar component can be styled as described in [Customizing the Appearance Using Styles and Skins](#). However, along with standard styling procedures, the calendar component has specific attributes that make styling instances of a calendar easier. These attributes are:

- `activityStyles`: Allows you to individually style each activity instance. For example, you may want to show activities belonging to different providers in different colors.
- `dateCustomizer`: Allows you to display strings other than the calendar date for the day in the month view. For example, you may want to display countdown or countup type numbers, as shown in [Figure 17-11](#). This attribute also allows you to add strings to the blank portion of the header for a day, for example to show the total number of hours worked per day. You can also use this attribute to color code certain days, such as holidays. See [How to Customize Dates](#).

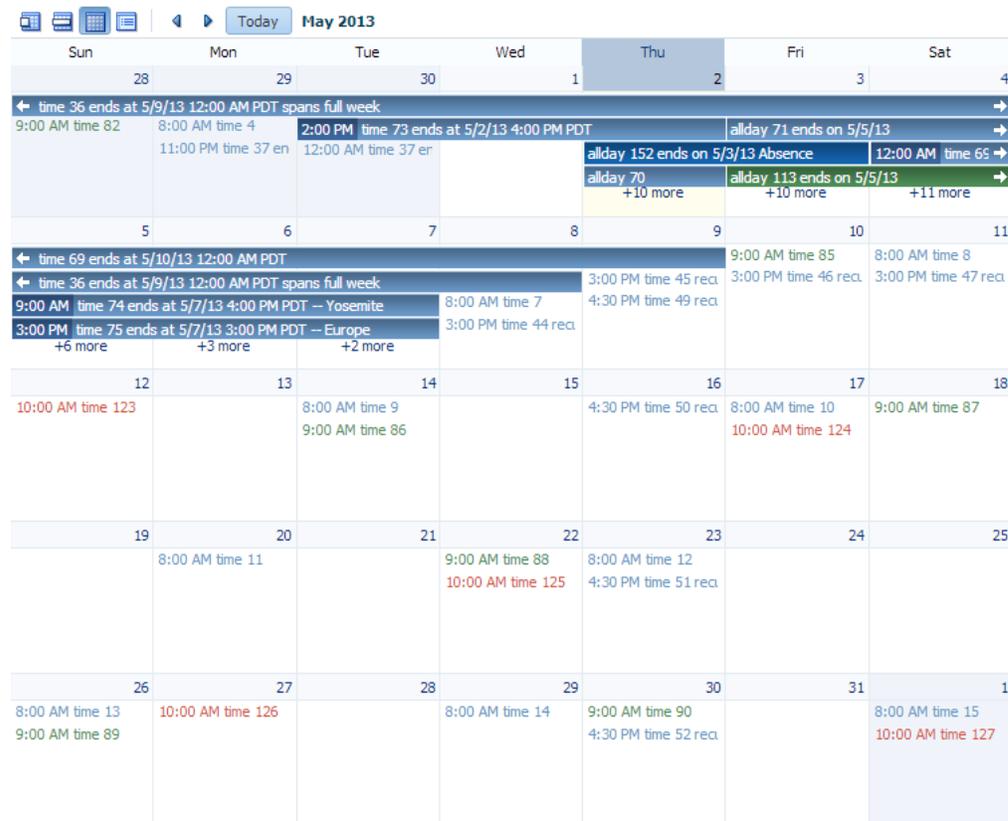
Figure 17-11 Customized Display of Dates in a Calendar

Sun	Mon	Tue	Wed	Thu	Fri	Sat
Week 18 -24	-23	-22	-21	-20	-19	-18
← time 36 ends at 5/9/13 12:00 AM PDT spans full week →						
9:00 AM time 82	8:00 AM time 4 11:00 PM time 37 en	2:00 PM time 73 ends at 5/2/13 4:00 PM PDT 12:00 AM time 37 er		+12 more	+12 more	+13 more
← time 36 ends at 5/9/13 12:00 AM PDT →						
← time 69 ends at 5/10/13 12:00 AM PDT →						
← time 36 ends at 5/9/13 12:00 AM PDT spans full week →						
+8 more	+5 more	+4 more	+2 more	3:00 PM time 45 rec. 4:30 PM time 49 rec.	9:00 AM time 85 3:00 PM time 46 rec.	8:00 AM time 8 3:00 PM time 47 rec.
Week 19 -17	-16	-15	-14	-13	-12	-11
Week 20 -10	-9	-8	-7	-6	-5	-4
10:00 AM time 123		8:00 AM time 9 9:00 AM time 86		4:30 PM time 50 rec.	8:00 AM time 10 10:00 AM time 124	9:00 AM time 87
Week 21 -3	-2	-1	0	+1	+2	+3
	8:00 AM time 11		9:00 AM time 88 10:00 AM time 125	8:00 AM time 12 4:30 PM time 51 rec.		
Week 22 +4	+5	+6	+7	+8	+9	+10
8:00 AM time 13 9:00 AM time 89	10:00 AM time 126		8:00 AM time 14	9:00 AM time 90 4:30 PM time 52 rec.		8:00 AM time 15 10:00 AM time 127

How to Style Activities

The `activityStyles` attribute uses `InstanceStyles` objects to style specific instances of an activity. The `InstanceStyles` class is a way to provide per-instance inline styles based on skinning keys.

The most common usage of the `activityStyles` attribute is to display activities belonging to a specific provider using a specific color. For example, the calendar shown in [Figure 17-12](#) shows activities belonging to three different providers. The user can change that color used to represent a provider's activities in the left panel. The `activityStyles` attribute is used to determine the color displayed for each activity, based on the provider with which it is associated.

Figure 17-12 Activities Styled to Display Color for Different Providers

Note that instead of using a single color, a range of a color is used in the calendar. This is called a color ramp. A color ramp is a set of colors in a color family and is used to represent the different states of activities. For example, Ted's activities use the blue color ramp. Activities whose time span is within one day are displayed in medium blue text. Activities that span across multiple days are shown in a medium blue box with white text. Darker blue is the background for the start time, while lighter blue is the background for the title. These three different blues are all part of the Blue color ramp.

The `CalendarActivityRamp` class is a subclass (of `InstanceStyles`) that supports some built-in color ramps and can take a representative color (for example, the blue chosen for Ted's activities) and return the correct color ramp to be used to display each activity in the calendar.

The `activityStyles` attribute must be bound to a `map` object. The map key is the set returned from the `getTags` method on an activity. The map value is an `InstanceStyles` object, most likely an instance of `CalendarActivityRamp`. This `InstanceStyles` object will take in skinning keys, and for each activity, styles will be returned.

Before you begin:

It may be helpful to have an understanding of calendar styles. See [Styling the Calendar](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To style activities:

1. In your `CalendarActivity` class, have the `getTags` method return a string set that will be used by the `activityStyles` attribute to map the returned string to a specific style. For example, to use the different color ramps for the different providers shown in [Figure 17-12](#), you must return a string for each provider. In this case, an activity belonging to the current user might return `Me`, an activity belonging to Mary might return `MJ`, and an activity belonging to Ted might return `TC`. For information about implementing the `CalendarActivity` class, see [How to Create a Calendar](#).
2. Create a map whose key is the set returned from the `getTags` method, and whose value is an `InstanceStyles` object (for example, a `CalendarActivityRamp` instance).

For example, to use the different color ramps shown in [Figure 17-12](#), you would create a map using the values shown in [Table 17-2](#).

Table 17-2 Map for activityStyles Attribute

Key (String Set)	Value (InstanceStyles Object)
<code>{"Me"}</code>	<code>CalendarActivityRamp.getActivityRamp(CalendarActivityRamp.RampKey.RED)</code>
<code>{"LE"}</code>	<code>CalendarActivityRamp.getActivityRamp(CalendarActivityRamp.RampKey.ORANGE)</code>
<code>{"TF"}</code>	<code>CalendarActivityRamp.getActivityRamp(CalendarActivityRamp.RampKey.BLUE)</code>

3. In the Structure window, select the calendar component, and in the Properties window, bind the `activityStyles` attribute to the map.

What Happens at Runtime: Activity Styling

During calendar rendering for each activity, the renderer calls the `CalendarActivity.getTags` method to get a string set. The string set is then passed to the map bound to the `activityStyles` attribute, and an `InstanceStyles` object is returned (which may be a `CalendarActivityRamp`).

Using the example:

- If the string set `{"Me"}` is passed in, the red `CalendarActivityRamp` is returned.
- If the string set `{"LE"}` is passed in, the orange `CalendarActivityRamp` is returned.
- If the string set `{"TF"}` is passed in, the blue `CalendarActivityRamp` is returned.

How to Customize Dates

If you want to display something other than the date number string in the day header of the monthly view, you can bind the `dateCustomizer` attribute to an implementation of a `DateCustomizer` class that determines what should be displayed for the date. You can use the `dateCustomizer` attribute to add strings to the blank portion of the header for a day, for example to show the total number of hours worked per day.

To color code dates, implement a new method named `getInlineStyle` in the `DateCustomizer` class. The `getInlineStyle` method returns inline CSS styles, which can be applied to the style attribute for the section of the calendar specified by a key. You can use `getInlineStyle` to set the background color on a date in the month grid. For accessibility, the information provided by the color coding must also be available to a screen reader. You can expose this information using the `dateHeaderStamp` facet to specify components that will be displayed in the header section of a date cell. For example, if the date is color coded because it is a holiday, you can specify an `af:image` with the `shortDesc` set to the holiday name, as well as programmatically specify which date it should display for, and the image will be displayed in the header for that date. For information about the `DateCustomizer` class, refer to the ADF Faces Javadoc. For information about the `dateHeaderStamp` facet, see the tag documentation for the calendar component.

Before you begin:

It may be helpful to have an understanding of calendar styling. See [Styling the Calendar](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for the Calendar](#).

To customize the date string:

1. Create a subclass of the `oracle.adf.view.rich.util.DateCustomizer` class. This subclass should determine what to display using the following skinning keys:

Keys passed to the `DateCustomizer.format` method:

- `af|calendar::day-header-row`: In day view, customize the day of the week in the header. For example, replace "Thursday" with "Thu".
- `af|calendar::day-header-row-misc`: In day view, customize the text beneath the day of the week in the header. For example, display "New Year's Day" on Jan 1.
- `af|calendar::list-day-of-month-link`: In list view, customize the text for the day of the month link. For example, replace "Jan 1" with "New Year's Day".
- `af|calendar::list-day-of-week-column`: In list view, customize the day of the week in the left list column. For example, replace "Thursday" with "Thu".
- `af|calendar::list-day-of-week-column-misc`: In list view, customize the text that appears beneath the day of the week in the left list column. For example, display "New Year's Day" on Jan 1.
- `af|calendar::month-grid-cell-header-day-link`: In month view, customize the date link labels in the cell header. For example, replace "5" with "-34".
- `af|calendar::month-grid-cell-header-misc`: In month view, add miscellaneous text to the empty area of the cell header. For example, on Jan 1, add the text "New Year's Day".
- `af|calendar::week-header-day-link`: In week view, customize the date link for each date in the header. For example, replace "Sun 1/1" with "New Year's Day".
- `af|calendar::week-header-cell-misc`: In week view, customize the text that appears beneath the date in the header. For example, display "New Year's Day" on Jan 1.

- `af|calendar::toolbar-display-range:day`: In day view, or in list view when `listType = day`, customize the date string on the toolbar.
- `af|calendar::toolbar-display-range:month`: In month view, or in list view when `listType = month`, customize the date string on the toolbar.

Keys passed to the `DateCustomizer.formatRange` method:

- `af|calendar::toolbar-display-range:week`: In week view, or in list view when `listType = week`, customize the date string on the toolbar.
- `af|calendar::toolbar-display-range:list`: In list view, or in list view when `listType = list`, customize the date string on the toolbar.

Keys passed to the `DateCustomizer.getInlineStyle` method:

- `af|calendar::list-row`: In list view, apply a CSS style to the current activity row.
- `af|calendar::month-grid-cell`: In month view, apply a CSS style for the specified date.

2. In a managed bean, create an instance of the `DateCustomizer` class. For example:

```
private DateCustomizer _dateCustomizer = new DemoDateCustomizer();
```

3. In the calendar component, bind the `dateCustomizer` attribute to the `DateCustomizer` instance created in the managed bean.

The following example shows how you can use the `DemoDateCustomizer` class to display the week number in the first day of the week, and a countdown number to a specific date instead of the day of the month, as shown in [Figure 17-11](#).

```
/* Date Customizer Displaying Countdown Numbers */
public class MyDateCustomizer extends DateCustomizer
{
    public String format(Date date, String key, Locale locale, TimeZone tz)
    {
        if ("af|calendar::month-grid-cell-header-misc".equals(key))
        {
            // return appropriate string
        }
        else if ("af|calendar::month-grid-cell-header-day-link".equals(key))
        {
            // return appropriate string
        }
        return null;
    }
}
```

The following code example shows how you can use the `DemoDateCustomizer` class to display color coded holidays.

```
/* Date Customizer Displaying Color Coded Holidays */
public class MyDateCustomizer extends DateCustomizer
{
    public String getInlineStyle(Date date, String key, Locale locale, TimeZone tz)
    {
        if ("af|calendar::day-all-day-activity-area".equals (key) ||
            "af|calendar::day-timed-activity-area".equals (key) ||
            "af|calendar::week-all-day-activity-area".equals (key) ||
```

```
        "af|calendar::week-timed-activity-area".equals (key) ||  
        "af|calendar::month-grid-cell".equals(key) ||  
        "af|calendar::list-row".equals(key))  
    {  
        Calendar curCal = Calendar.getInstance (tz, locale);  
        curCal.setTime (date);  
  
        if (_getUSHoliday (curCal) != null)  
            return "background-color: #fafaeb;";  
    }  
    return null;  
}
```

The following code example shows how you can use the `DemoDateCustomizer` class to display color coded holidays using the `dateHeaderStamp` facet as it is used on the `dateCustomizerCalendar.jspx` page of the File Explorer application.

```
/* Date Customizer Displaying Color Coded Holidays with dateHeaderStamp */  
<af:calendar id="cal" ..>  
    <f:facet name="dateHeaderStamp">  
        <af:image rendered="{calendarBean.dateCustomizer.USHoliday}"  
            source="/images/holidayStar_16X16.png"  
            shortDesc="{calendarBean.dateCustomizer.dateHeaderDesc}"  
            ../>  
    </f:facet>  
</af:calendar>
```

18

Using Output Components

This chapter describes how to use the ADF Faces `outputText`, `outputFormatted`, `image`, `icon`, and `statusIndicator` components to display output text, images, and icons, and how to use the `media` component to enable users to play video and audio clips.

This chapter includes the following sections:

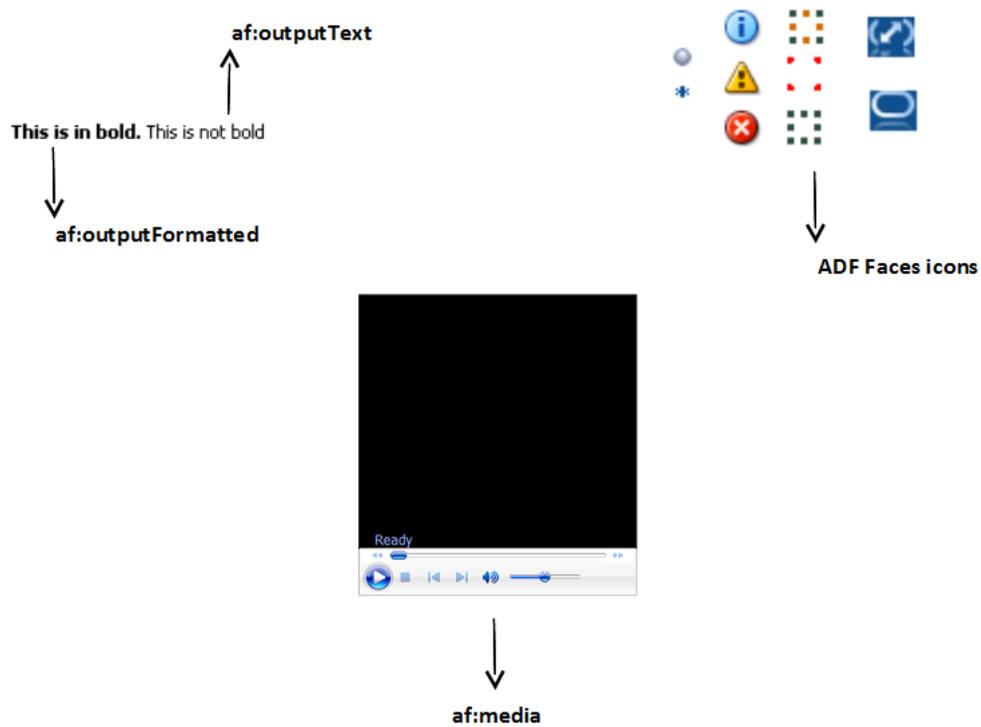
- [About Output Text, Image, Icon, and Media Components](#)
- [Displaying Output Text and Formatted Output Text](#)
- [Displaying Icons](#)
- [Displaying Images](#)
- [Using Images as Links](#)
- [Displaying Application Status Using Icons](#)
- [Playing Video and Audio Clips](#)

About Output Text, Image, Icon, and Media Components

ADF Faces has components that you can use to format the output text, images, icons, and use the media components to enable users to play video and audio clips.

ADF Faces provides components for displaying text, icons, and images, and for playing audio and video clips on JSF pages.

Figure 18-1 ADF Faces Output Components



Output Components Use Case and Examples

The `outputText` component can be used as a child to many other components to display read-only text. When you need the text to be formatted, you can use the `outputFormatted` component. For example, you may want to use bold formatted text within instruction text, as shown in [Figure 18-2](#).

Figure 18-2 Use the `outputFormatted` Component in Instruction Text

Speak with a FileExplorer.com Customer Service Representative

We're available 24 hours a day, 7 days a week, 365 days a year.
Let us know a good time to call you, and we'll have a customer service representative contact you.

Many ADF Faces components can have icons associated with them. For example, in a menu, each of the menu items can have an associated icon. You identify the image to use for each one as the value of an `icon` attribute for the menu item component itself. Information and instructions for adding icons to components that support them are covered in those components' chapters. In addition to providing icons within components, ADF Faces also provides icons used when displaying messages. You can use these icons outside of messages as well.

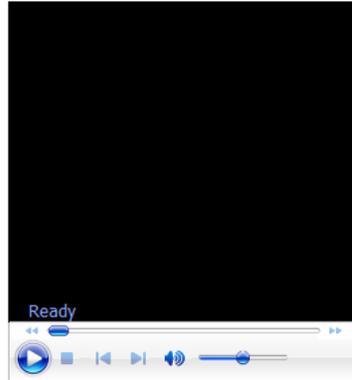
To display an image on a page, you use the `image` component. Images can also be used as links (including image maps) or to depict the status of the server.

The `media` component is used to display audio-visual content, such as advertisements or directions to complete a task. The `media` component can be configured to display all

controls, typical controls, minimal controls, no visible controls (for example, controls are available only from a context menu), or no controls at all. Typically, you would not display controls when the clip is very short and control is not needed.

When the `media` component is the primary component on the page, then typically all controls are displayed, as shown in [Figure 18-3](#).

Figure 18-3 All Controls are Displayed at the Bottom of the Player



Additional Functionality for Output Components

You may find it helpful to understand other ADF Faces features before you implement your output components. Additionally, once you have added these components to your page, you may find that you need to add functionality such as drag and drop and accessibility. Following are links to other functionality that output components can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Conversion:** In some cases, a value may need to be converted to a string in order to display. For information about conversion, see [Validating and Converting Input](#).
- **Drag and drop:** You can configure a page so that a user can drag and drop output components or values of output components, to another area on a page. See [Adding Drag and Drop Functionality](#).
- **Localization:** Instead of entering values for attributes that take strings as values, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).
- **Skins:** You can change the look and feel of output components by changing the skin. See [Customizing the Appearance Using Styles and Skins](#).

Displaying Output Text and Formatted Output Text

ADF Faces provides `outputText` and `outputFormatted` components that you can use to display unformatted and a limited range of formatted text to the users.

There are two ADF Faces components specifically for displaying output text on pages: `outputText`, which displays unformatted text, and `outputFormatted`, which displays text and can include a limited range of formatting options.

To display simple text specified either explicitly or from a resource bundle or bean, use the `outputText` component. You define the text to be displayed as the value of the `value` attribute. For example:

```
<af:outputText value="The submitted value was: "/>
```

The following example shows two `outputText` components: the first specifies the text to be displayed explicitly, and the second takes the text from a managed bean and converts the value to a text value ready to be displayed (for information about conversion, see [Adding Conversion](#)).

```
<af:panelGroupLayout>
  <af:outputText value="The submitted value was: "/>
  <af:outputText value="#{demoInput.date}">
    <af:convertDateTime dateStyle="long"/>
  </af:outputText>
</af:panelGroupLayout>
```

You can use the `escape` attribute to specify whether or not special HTML and XML characters are escaped for the current markup language. By default, characters are escaped.

The following example illustrates two `outputText` components, the first of which uses the default value of `true` for the `escape` attribute, and the second of which has the attribute set to `false`.

```
<af:outputText value="&lt;h3>output &amp; heading&lt;/h3>" />
<af:outputText value="&lt;h3>output &amp; heading&lt;/h3>"
  escape="false" />
```



Note:

Avoid setting the `escape` attribute to `false` unless absolutely necessary. When `escape` is set to `false`, your website may be exposed to cross-site scripting attacks if the value of the `outputText` component is in any way derived from values supplied by a user. A better option is to use the `outputFormatted` component, which allows a limited number of HTML tags. In addition, nearly all attributes are ignored when the `escape` attribute is set to `false` (for example, `styleClass` is not output).

Figure 18-4 shows the different effects of the two different settings of the `escape` attribute when viewed in a browser.

Figure 18-4 Using the escape Attribute for Output Text

```
<h3>output & heading</h3>
output & heading
```

As with the `outputText` component, the `outputFormatted` component also displays the text specified for the `value` attribute, but the value can contain HTML tags. Use the formatting features of the `outputFormatted` component specifically when you want to

format only parts of the value in a certain way. If you want to use the same styling for the whole component value, instead of using HTML within the value, apply a style to the whole component. If you want all instances of a component to be formatted a certain way, then you should create a custom skin. For information about using inline styles and creating skins, see [Customizing the Appearance Using Styles and Skins](#).

The following example shows an `outputFormatted` component displaying only a few words of its value in bold (note that you need to use entities such as `<` on JSPX pages).

```
<af:outputFormatted value="&lt;b>This is in bold.&lt;/b> This is not bold"/>  
<af:outputFormatted value="<b>This is in bold.</b> This is not bold"/>
```

Figure 18-5 shows how the component displays the text.

Figure 18-5 Text Formatted Using the `outputFormatted` Component

This is in bold. This is not bold

How to Display Output Text

Before displaying any output text, decide whether or not any parts of the value must be formatted in a special way. If they do, then use an `outputFormatted` component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Output Text and Formatted Output Text](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To display output text:

1. In the Components window, from the Text and Selection panel, drag and drop an **Output Text** onto the page. To create an `outputFormatted` component, drag and drop an **Output Text (Formatted)** from the Components window.

 **Tip:**

If parts of the value require special formatting, use an `outputFormatted` component.

 **Tip:**

If you plan to support changing the text of the component through active data (for example, data being pushed from the data source will determine the text that is displayed), then you should use the `activeOutputText` component instead of the `outputText` component. Create an `activeOutputText` component by dragging an **Output Text (Active)** from the Components window.

- Expand the Common section of the Properties window and set the **value** attribute to the value to be displayed. If you are using the `outputFormatted` component, use HTML formatting codes to format the text as needed, as described in [Table 18-1](#) and [Table 18-2](#).

The `outputFormatted` component also supports the `styleUsage` attribute whose values are the following predefined styles for the text:

- `inContextBranding`
- `instruction`
- `pageStamp`

[Figure 18-6](#) shows how the `styleUsage` values apply styles to the component.

Figure 18-6 styleUsage Attribute Values

This text has no StyleUsage set
 This is the `inContextBranding` style
 This is the `instruction` style
 This is the `pageStamp` style

 **Note:**

If the `styleUsage` and `styleClass` attributes are both set, the `styleClass` attribute takes precedence.

What You May Need to Know About Allowed Format and Character Codes in the `outputFormatted` Component

Only certain formatting and character codes can be used. [Table 18-1](#) lists the formatting codes allowed for formatting values in the `outputFormatted` component.

Table 18-1 Formatting Codes for Use in `af:outputFormatted` Values

Formatting Code	Effect
<code>
</code>	Line break
<code><hr></code>	Horizontal rule

Table 18-1 (Cont.) Formatting Codes for Use in af:outputFormatted Values

Formatting Code	Effect
<code>...</code> <code>...</code> <code>...</code>	Lists: ordered list, unordered list, and list item
<code><h1>...</h1></code> to <code><h6>...</h6></code>	Headings: Ranked from <code><h1></code> , the most important heading, down to <code><h6></code> , the least important heading
<code><p>...</p></code>	Paragraph
<code>...</code>	Bold
<code><i>...</i></code>	Italic
<code><tt>...</tt></code>	Teletype or monospaced
<code><big>...</big></code>	Larger font
<code><small>...</small></code>	Smaller font
<code><pre>...</pre></code>	Preformatted: layout defined by whitespace and line break characters preserved
<code>...</code>	Span the enclosed text
<code><a>...</code>	Anchor

[Table 18-2](#) lists the character codes for displaying special characters in the values.

Table 18-2 Character Codes for Use in af:outputFormatted Values

Character Code	Character
<code>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>&amp;</code>	Ampersand
<code>&reg;</code>	Registered
<code>&copy;</code>	Copyright
<code>&#160;</code>	Nonbreaking space
<code>&quot;</code>	Double quotation marks

The attributes `class`, `style`, and `size` can also be used in the `value` attribute of the `outputFormatted` component, as can `href` constructions. All other HTML tags are ignored.

 **Note:**

For security reasons, JavaScript is not supported in output values.

 **Note:**

Because other components (like `panelHeader`) can also display `<h1>`-`<h6>` tags, check the rendered HTML to ensure that any hardcoded header tags in your `outputFormatted` component display in relative order to those added by other components.

Displaying Icons

ADF Faces provides ready to use icons with message components. You can also use these icons outside of a message component.

ADF Faces provides a set of icons used with message components, shown in [Figure 18-7](#).

Figure 18-7 ADF Faces Icons



If you want to display icons outside of a `message` component, you use the `icon` component and provide the name of the icon type you want to display.

 **Note:**

The images used for the icons are determined by the skin the application uses. If you want to change the image, create a custom skin. See [Customizing the Appearance Using Styles and Skins](#).

How to Display Icons

When you use messages in an ADF Faces application, the icons are automatically added for you. You do not have to add them to the `message` component. However, you can also use the icons outside of a `message` component. To display one of the standard icons defined in the skin for your application, you use the `icon` component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Icons](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To display a standard icon:

1. In the Components window, from the General Controls panel, drag and drop an **Icon** onto the page.
2. Expand the Common section and set **Name** to the name of one of the icon functions shown in [Figure 18-7](#). For example, if you want to display a red circle with a white X, you would set **Name** to `error`.
3. Expand the Appearance section and set **ShortDesc** to the text you want to be displayed as the alternate text for the icon.

Displaying Images

The ADF Faces image component allows you to display an image stored locally. You must provide the location using the `source` attribute.

To display an image on a page, you use the `image` component and set the `source` attribute to the URI where the file is located. The `image` component also supports accessibility description text by providing a way to link to a long description of the image.

The `image` component can also be used as a link and can include an image map, but it must be placed inside a `link` component. See [Using Images as Links](#).

How to Display Images

You use the `image` component to display images.

Before you begin:

You may find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To display an image:

1. In the Components window, from the General Controls panel, drag and drop an **Image** onto the page.

Tip:

If you plan to support changing the `source` attribute of the image through active data (for example, data being pushed from the data source will determine the image that is displayed), then you should use the `activeImage` component instead of the `image` component. Create an `activeImage` component by dragging an **Image (Active)** from the Components window.

2. In the Insert Image dialog, set the following:
 - **Source:** Enter the URI to the image file.
 - **ShortDesc:** Set to the text to be used as the alternate text for the image.
3. If you want to include a longer description for the image, in the Properties window, set **LongDescURL** attribute to the URI where the information is located.

Using Images as Links

Using the ADF Faces link component you can render an image as a link to one or more destinations. Users can then click on the image to go to the destination link.

ADF Faces provides the `link` component, which can render an image as a link, along with optional text. You can set different icons for when the user hovers the mouse over the icon, and for when the icon is depressed or disabled. For information about the `link` component, see [Using Buttons and Links for Navigation](#).

You can use an image as a `link` component to one or more destinations. If you want to use an image as a simple link to a single destination, use a `link` component to enclose your image, and set the `destination` attribute of the `link` component to the URI of the destination for the link.

If your image is being used as a graphical navigation menu, with different areas of the graphic navigating to different URIs, enclose the `image` component in a `link` component and create a server-side image map for the image.

How to Use Images as Links

You use the `link` component to render an image as a link.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Using Images as Links](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To use an image as one or more Link components:

1. In the Components window, from the General Controls panel, drag and drop a **Link** onto the page.
2. Drag and drop an **Image** as a child to the **Link** component.
3. In the Insert Image dialog, set the following:
 - **Source**: Enter the URI to the image file.
 - **ShortDesc**: Set to the text to be used as the alternate text for the image.
4. If different areas of the image are to link to different destinations:
 - Create an image map for the image and save it to the server.
 - In the Properties window, set the **ImageMapType** attribute to **server**.
 - Select the **Link** component and in the Properties window, set **Destination** to the URI of the image map on the server.
5. If the whole image is to link to a single destination, select the **Link** component and enter the URI of the destination as the value of **Destination**.

Displaying Application Status Using Icons

The ADF Faces `statusIndicator` component allows you to display the server status through animated icons to users.

ADF Faces provides the `statusIndicator` component, which you can use to indicate server activity. What displays depends both on the skin your application uses and on how your server is configured. By default, the following are displayed:

- When your application is configured to use the standard data transfer service, during data transfer an animated spinning icon is displayed:



When the server is not busy, a static icon is displayed:



- When your application is configured to use the Active Data Service (ADS), what the status indicator displays depends on how ADS is configured.

 **Note:**

ADS allows you to bind your application to an active data source. You must use the Fusion technology stack in order to use ADS. See Using the Active Data Service in *Developing Fusion Web Applications with Oracle Application Development Framework*.

ADS can be configured to either have data pushed to the model, or it can be configured to have the application poll for the data at specified intervals. [Table 18-3](#) shows the icons that are used to display server states for push and poll modes (note that the icons are actually animated).

Table 18-3 Icons Used in Status Indicator for ADS

Icon	Push Mode	Pull Mode
	At the first attempt at connecting to the server.	At the first attempt at connecting to server.

Table 18-3 (Cont.) Icons Used in Status Indicator for ADS

Icon	Push Mode	Pull Mode
	When the first connection is successfully established.	When the first connection is successfully established and when a connection is reestablished.
	When subsequent attempts are made to reconnect to the server.	Before every poll request.
	When a connection cannot be established or reestablished.	When the configured number of poll attempts are unsuccessful.

After you drop a `statusIndicator` component onto the page, you can use skins to change the actual image files used in the component. For information about using skins, see [Customizing the Appearance Using Styles and Skins](#).

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Application Status Using Icons](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To use the status indicator icon:

1. In the Components window, from the General Controls panel, drag and drop a **Status Indicator** onto the page.
2. Use the Properties window to set any needed attributes.

 **Tip:**

For help in setting attributes, use the field's dropdown menu to view a description of the attribute.

Playing Video and Audio Clips

The ADF Faces media component supports audio and video clips to be added on the application pages. You can also specify the preferred type of media player that users can opt to play the clip.

The ADF Faces `media` component allows you to include video and audio clips on your application pages.

The media control handles two complex aspects of cross-platform media display: determining the best player to display the media, and sizing the media player.

You can specify which media player is preferred for each clip, along with the size of the player to be displayed for the user. By default, ADF Faces uses the MIME type of the media resource to determine the best media player and the default inner player size to use, although you can specify the type of content yourself, using the `contentType` attribute.

You can specify which controls are to be available to the user, and other player features such as whether or not the clip should play automatically, and whether or not it should play continuously or a specified number of times.

How to Allow Playing of Audio and Video Clips

Once you add a `media` component to your page, you can configure which media player to use by default, the size of the player and screen, the controls, and whether or not the clip should replay.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Playing Video and Audio Clips](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Output Components](#).

To include an audio or video clip in your application page:

1. In the Components window, from the General Controls panel, drag and drop a **Media** onto the page.
2. In the Insert Media dialog, set the following attributes:
 - **Source:** Enter the URI to the media to be played.
 - **StandbyText:** Enter a message that will be displayed while the content is loading.
3. Expand the Common section of the Properties window and set the following:
 - **Player:** Select the media player that should be used by default to play the clip. You can choose from Real Player, Windows Media Player, or Apple Quick Time Player.

Alternatively, you can create a link in the page that starts the playing of the media resource based on the user agent's built-in content type mapping. The media control attempts to pick the appropriate media player using the following steps:

- If the primary MIME type of the content is image, the built-in user-agent support is used.
- If a media player has been specified by the `player` attribute, and that player is available on the user agent and can display the media resource, that player is used.
- If one player is especially good at playing the media resource and that player is available on the user agent, that player is used.
- If one player is especially dominant on the user agent and that player can play the media resource, that player is used.

- The player connected to the link provided on the page is used.
 - **Autostart:** Set to **True** if you want the clip to begin playing as soon as it loads.
 - **ContentType:** Enter the MIME type of the media to play. This will be used to determine which player to use, the configuration of the controls, and the size of the display.
4. Expand the Appearance section of the Properties window and set the following:
- **Controls:** Select the amount and types of controls you want the player to display.

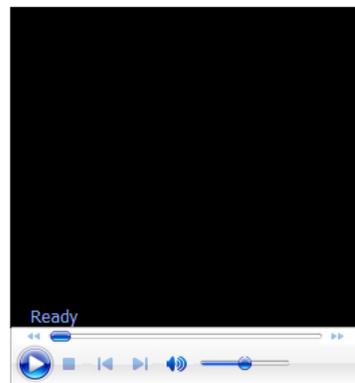
Because the set of controls available varies between players, you define what set of controls to display in a general way, rather than listing actual controls. For example, you can have the player display all controls available, the most commonly used controls, or no controls.

The following example uses the `all` setting for a `media` component.

```
<af:media source="/images/myvideo.wmv" controls="all"/>
```

Figure 18-8 shows how the player is displayed to the user.

Figure 18-8 Media Player with All Controls



The following values are valid:

- **All:** Show all available controls for playing media on the media player.
Using this setting can cause a large amount of additional space to be required, depending on the media player used.
- **Minimal:** Show a minimal set of controls for playing media on the media player.
This value gives users control over the most important media playing controls, while occupying the least amount of additional space on the user agent.
- **None:** Do not show any controls for the media player and do not allow control access through other means, such as context menus.
You would typically use this setting only for kiosk-type applications, where no user control over the playing of the media is allowed. This setting is typically used in conjunction with settings that automatically start the playback, and to play back continuously.

- **NoneVisible:** Do not show any controls for the media player, but allow control access through alternate means, such as context menus.

You would typically use this value only in applications where user control over the playing of the media is allowed, but not encouraged. As with the `none` setting, this setting is typically used in conjunction with settings that automatically start the playback, and to play back continuously.

- **Typical:** Show the typical set of controls for playing media on the media player.

This value, the default, gives users control over the most common media playing controls, without occupying an inordinate amount of extra space on the user agent.

- **Width and Height:** Define the size in pixels of the complete display, including the whole player area, which includes the media content area.

 **Tip:**

Using the `width` and `height` attributes can lead to unexpected results because it is difficult to define a suitable width and height to use across different players and different player control configurations. Instead of defining the size of the complete display, you can instead define just the size of the media content area using the `innerWidth` and `innerHeight` attributes.

- **InnerWidth and InnerHeight:** Define the size in pixels of only the media content area. This is the preferred scheme, because you control the amount of space allocated to the player area for your clip.

 **Tip:**

If you do not specify a size for the media control, a default inner size, determined by the content type of the media resource, is used. While this works well for audio content, it can cause video content to be clipped or to occupy too much space.

If you specify dimensions from both schemes, such as a `height` and an `innerHeight`, the overall size defined by the `height` attribute is used. Similarly, if you specify both a `width` and an `innerWidth`, the `width` attribute is used.

5. Expand the Behavior section and set **Autostart**. By default, playback of a clip will not start until the user starts it using the displayed controls. You can specify that playback is to start as soon as the clip is loaded by setting the `autostart` attribute to `true`.

Set **PlayCount** to the number of times you want the media to play. Once started, by default, the clip will play through once only. If the users have controls available, they can replay the clip. However, you can specify that the clip is to play back a fixed number of times, or loop continuously, by setting a value for the `playCount` attribute. Setting the `playCount` attribute to 0 replays the clip continuously. Setting the attribute to some other number plays the clip the specified number of times.

The following example shows an `af:media` component in the source of a page. The component will play a video clip starting as soon as it is loaded and will continue to play the clip until stopped by the user. The player will display all the available controls.

```
<af:media source="/components/images/seattle.wmv" playCount="0"  
    autostart="true" controls="all"  
    innerHeight="112" innerWidth="260"  
    shortDesc="My Video Clip"  
    standbyText="My video clip is loading"/>
```

19

Displaying Tips, Messages, and Help

This chapter describes how to use the `shortDesc` attribute to display tips, and how to display messages for ADF Faces components. This chapter also describes how to provide different levels of help information for users.

This chapter includes the following sections:

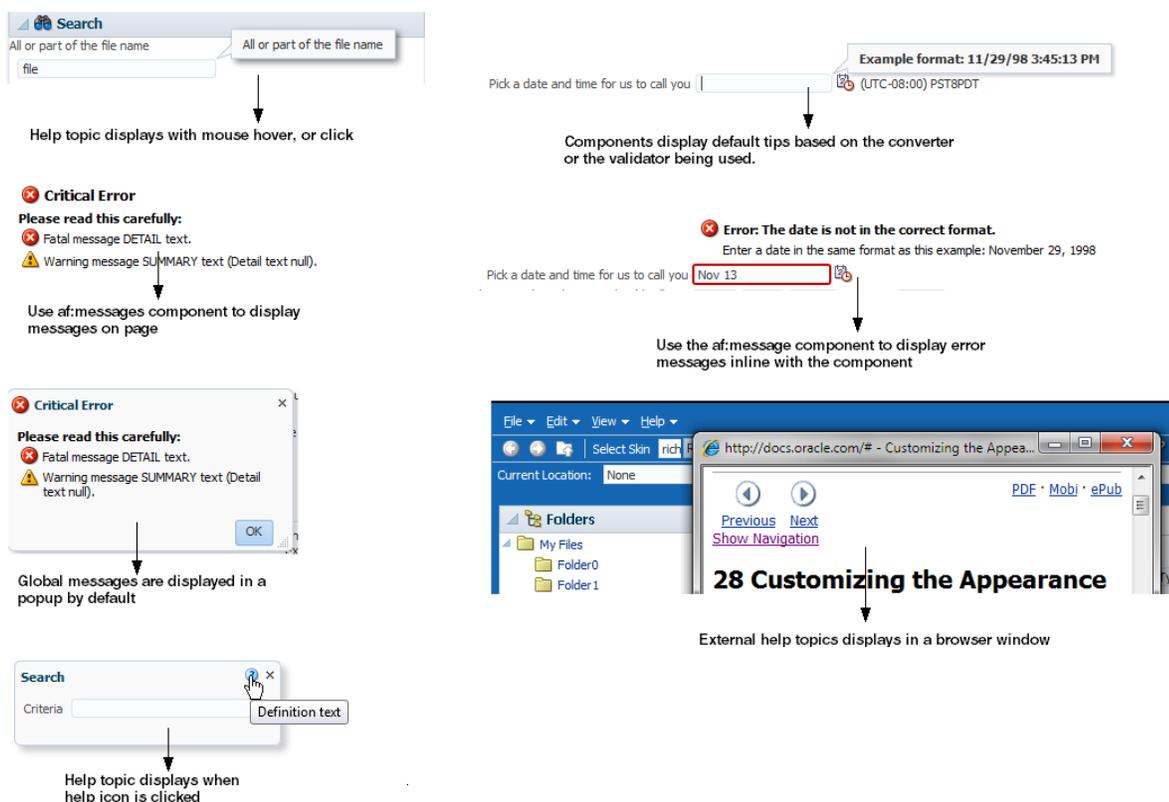
- [About Displaying Tips and Messages](#)
- [Displaying Tips for Components](#)
- [Displaying Hints and Error Messages for Validation and Conversion](#)
- [Grouping Components with a Single Label and Message](#)
- [Displaying Help for Components](#)
- [Combining Different Message Types](#)

About Displaying Tips and Messages

ADF Faces has many components that you can use to display informational text in an application. For example, using `shortDesc` attribute you can display tip information when an user hovers the cursor, using `inputText` component you can display popup messages to users, and so on.

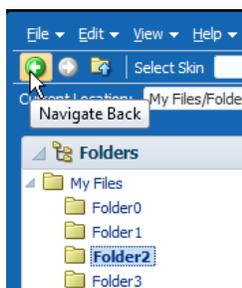
ADF Faces provides many different ways for displaying informational text in an application. You can create simple tip text, validation and conversion tip text, validation and conversion failure messages, as well as elaborate help systems.

Figure 19-1 ADF Messaging Components



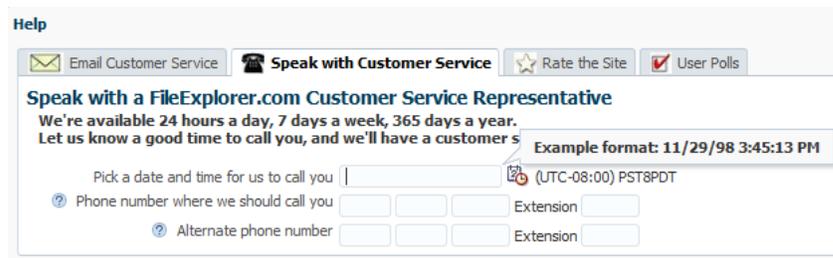
Many ADF Faces components support the `shortDesc` attribute, which for most components, displays tip information when a user hovers the cursor over the component. Figure 19-2 shows a tip configured for a toolbar button.

Figure 19-2 Tip Displays Information on Mouse Hover



Along with tips, some ADF Faces components (such as the `inputText` component, or the selection components) can display popup hint messages and failure messages used for validation and conversion. You can create two types of messages: component level messages that apply to a specific component, and global-level messages that apply to a page or group of components. Figure 19-3 shows a hint message configured for an Input Date field.

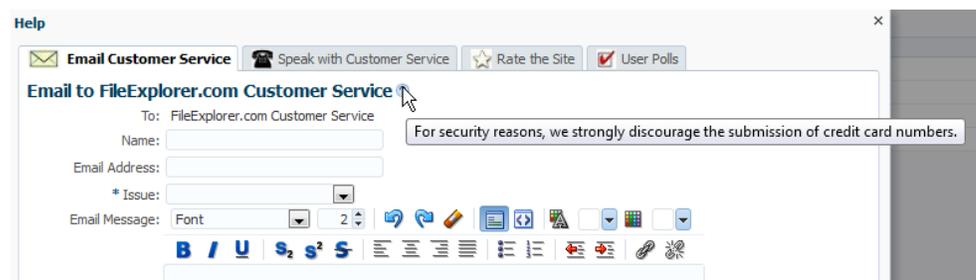
Figure 19-3 Hint Message Displays Format of Input Date



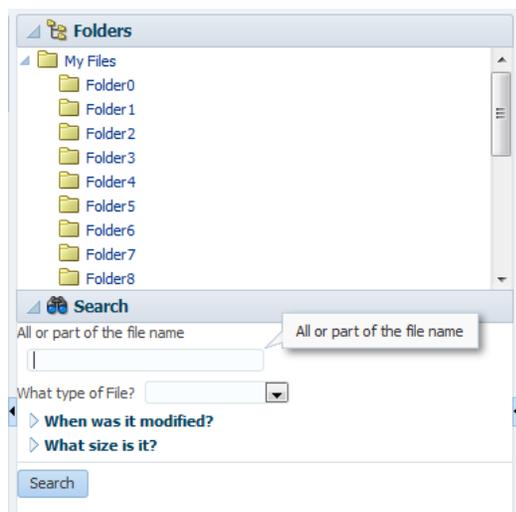
Along with configuring messages for individual component instances, you can create a separate help system that provides information that can be reused throughout the application. You create help information using different types of providers, and then reference the help text from the UI components. The following are the three types of help supported by ADF Faces:

- Definition: Provides a help icon (question mark in a blue circle) with the help text appearing when the user mouses over the icon, as shown in [Figure 19-4](#).

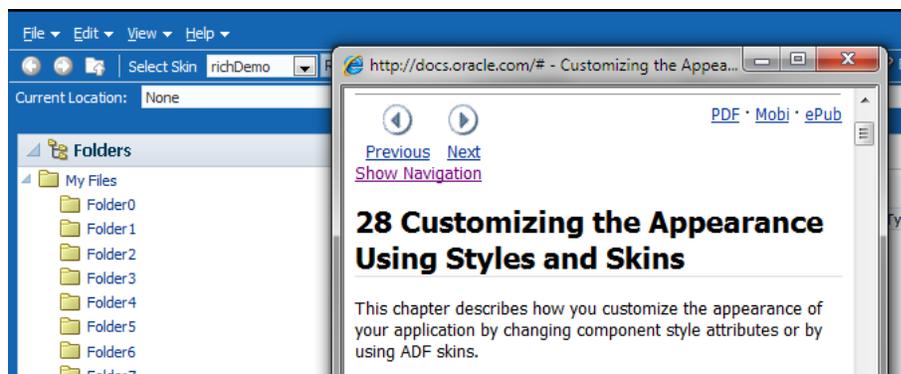
Figure 19-4 Definition Messages Display When Mousing Over the Icon



- Instruction: Depending on the component, this type of help either provides instruction text within the component (as with `panelHeader` components), or displays text in the note window that is opened when the user clicks in the component, as shown in [Figure 19-5](#). The text can be of any length.

Figure 19-5 Instruction Messages Display in a Note Window

- **External URL:** You can have a help topic that resides in an external application, which will open in a separate browser window. For example, instead of displaying instruction help, [Figure 19-6](#) shows the Select Skin `selectOneChoice` component configured to open a help topic about skins. When a user clicks the `selectOneChoice` component, the help topic opens.

Figure 19-6 External URL Help Opens in a New Window

For information about creating help systems, see [Displaying Help for Components](#).

Messaging Components Use Cases and Examples

Messages can typically be divided into two types: error messages that display when an error occurs in the application, for example when a user enters incompatible information, and informational messages that provide for example, hints for using a component or for completing a task on a page.

Informational messages can range from simple tooltips to comprehensive help systems. Tooltips should be used when the component for which you want to display hints or information does not support help text. However, tooltip text must be very brief. If you have to display more detailed information, or if the text can be reused among many component instances, consider using help text instead.

You create tooltips by configuring the `shortDesc` attribute on a component. The value of that attribute then displays in a note window when the user hovers over the component or clicks the component (such as `inputText` component), as shown in [Figure 19-7](#).

Figure 19-7 Tooltip for a Component



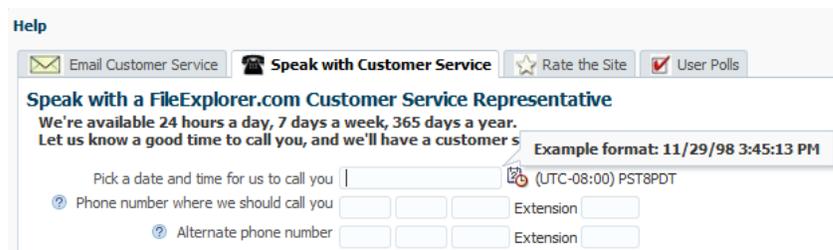
For information about tooltips, see [Displaying Tips for Components](#).

Error messages use the JSF messaging API. There are two types of error messages: component messages where the message applies to the specific component only, and global messages, where the message applies to more than one component or the whole page.

By default, the `noteWindow` component is used for component error messages. When you configure conversion or validation on any input component, validation and conversion hints and errors are automatically displayed in the `noteWindow` component. You do not need to add the component to the page.

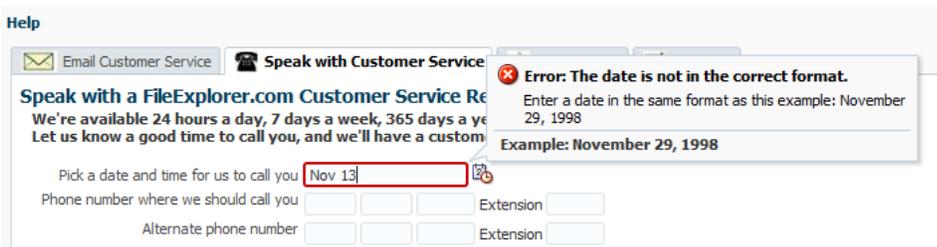
For example, when users click **Help > Give Feedback** in the File Explorer application, a dialog displays where they can enter a time and date for a customer service representative to call. Because the `inputDate` component contains a converter, when the user clicks in the field, a note window displays a hint that shows the expected pattern, as shown in [Figure 19-8](#). If the `inputDate` component was also configured with a minimum or maximum value, the hint would display that information as well. These hints are provided by the converters and validators automatically.

Figure 19-8 Attached Converters and Validators Include Messages



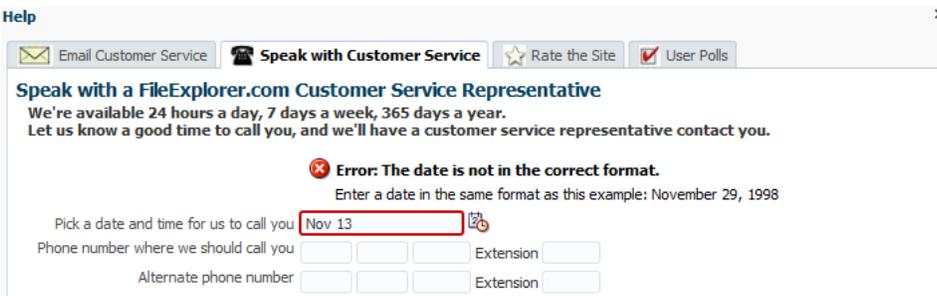
If a user enters a date incorrectly in the field shown in [Figure 19-8](#), an error message is displayed, as shown in [Figure 19-9](#). Note that the error message appears in the note window along with the hint.

Figure 19-9 Validation and Conversion Errors Display in Note Window



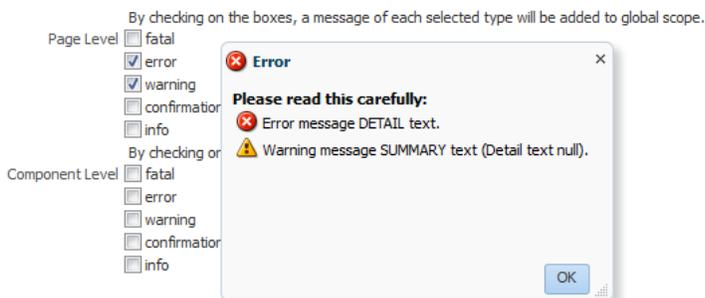
If you want to display an error message for a non-ADF Faces component, or if you want the message to be displayed inline instead of the note window, use the ADF Faces message component. When you use this component, messages are displayed next to the component, as shown in [Figure 19-10](#).

Figure 19-10 Use the message Component to Display Messages Inline



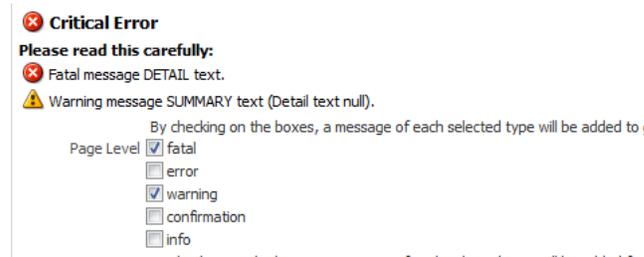
Global messages are by default displayed in a dialog, as shown in [Figure 19-11](#). You do not need to add the popup component to the page.

Figure 19-11 Global Messages Display in a Popup Dialog



If instead you want the error messages to display directly on the page, use the messages component. When you use this component, the messages are displayed in a list at the top of the page, as shown in [Figure 19-12](#).

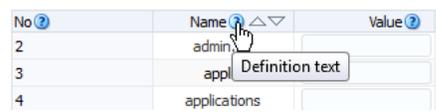
Figure 19-12 Use the messages Component to Display Global Messages on the Page



For information about error messages, see [Displaying Hints and Error Messages for Validation and Conversion](#).

When you want to display more information that can fit in a tooltip, use definition help. When you configure definition help for most components, a help icon is displayed next to the component. The help text is displayed when the mouse hovers over the component, as shown in [Figure 19-13](#).

Figure 19-13 Definition Help for a Column Component



For information about definition help, see [Displaying Help for Components](#).

When you want to display field-level help, configure an input component to use instruction text. When the user clicks in the component, the help text is displayed in a note window, as shown in [Figure 19-14](#).

Figure 19-14 Instruction Text for a Component



When you want to display instructions for a task, configure instruction help for a container component. The text will appear in the header of the component, as shown in [Figure 19-15](#).

Figure 19-15 Instruction Text for the panelHeader Component

Best Practice:

Instruction text for input components should be used only when the typical user may fail to perform a task without assistance. Excessive use of instruction text clutters the page with directions or distracts users with note windows that may also obscure related page elements.

When you need to provide comprehensive help, you can use the help icon to link to an external help system available through a URL.

For information about instruction and external help, see [Displaying Help for Components](#).

Additional Functionality for Message Components

You may find it helpful to understand other ADF Faces features before you implement your message components and help functionality. Additionally, once you have added these components to your page, you may find that you need to add functionality such as skinning to change icons and accessibility and using resource bundles to store message text. Following are links to other functionality that message components can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Client events:** If you want your help topic to launch using JavaScript, you use a listener for a client event. For information about client-side events, see [Using JavaScript for ADF Faces Client Events](#).
- **Skinning:** The icons displayed for messages and help are determined by the skin used by the application. You can change the icons by creating a new skin. See [Customizing the Appearance Using Styles and Skins](#).
- **Localization:** Instead of directly entering text for messages, you can use property files. These files allow you to manage translation of these strings. See [Internationalizing and Localizing Pages](#).

Displaying Tips for Components

ADF Faces components has the `shortDesc` attribute that you can use to display short text as a tip, a note, a popup message, and so on. Users can see this text when they hover the cursor over an `inputText` component.

ADF Faces components use the `shortDesc` attribute to display a tip when the user hovers the mouse over the component. Input components display the tips in their note window. Other component types display the tip in a standard tip box. This text should be kept short. If you have to display more detailed information, or if the text can be reused among many component instances, consider using help text, as described in [Displaying Help for Components](#).

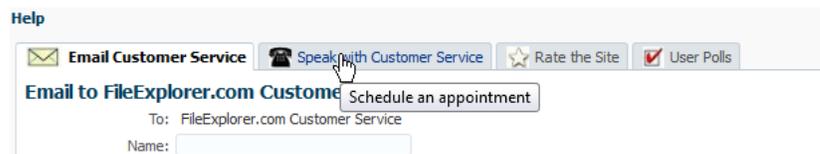
Figure 19-16 shows the effect when the cursor hovers over an `inputText` component.

Figure 19-16 Tip for an `inputText` Component



Figure 19-17 shows a tip as displayed for a `showDetailItem` component.

Figure 19-17 Tip for a `showDetailItem` Component



How to Display Tips for Components

You use the `shortDesc` attribute on a component to display a tip.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Tips for Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Message Components](#).

To define a tip for a component:

1. In the Structure window, select the component for which you want to display the tip.
2. In the Properties window, expand the **Appearance** section and enter a value for the `shortDesc` attribute.

Tip:

The value should be less than 80 characters, as some browsers will truncate the tip if it exceeds that length.

If the text to be used is stored in a resource bundle, use the dropdown list to select **Select Text Resource**. Use the Select Text Resource dialog to either search for appropriate text in an existing bundle, or to create a new entry in an existing bundle. For information about using resource bundles, see [Internationalizing and Localizing Pages](#).

Displaying Hints and Error Messages for Validation and Conversion

In ADF Faces, when you allow users to validate a value or a value is to be converted, default error messages and tips for using the appropriate values are displayed. However, you can override the default hint and error messages for the validators and converters.

Validators and converters have a default hint that is displayed to users when they click in the associated field. The default hint automatically displays in a note window. For converters, the hint usually tells the user the correct format to use. For validators, the hint is used to convey what values are valid.

For example, in the File Explorer application, when a user clicks in the input date field on the Speak with Customer Service page, a tip is displayed showing the correct format to use, as shown in [Figure 19-18](#).

Note:

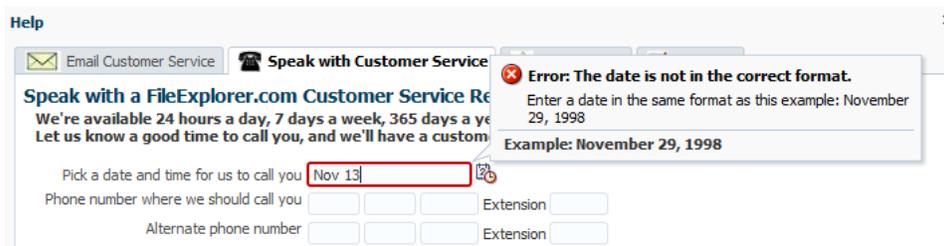
All the hints across the input fields can be turned off by setting the context parameter `oracle.adf.view.rich.component.DEFAULT_HINT_DISPLAY` to `none`. By default, this parameter is set to `auto` allowing the framework to decide the presentation mode for component hint. The default behavior is to show the hint as a note window tip on the component when it receives the focus.

Figure 19-18 Validators and Converters Have Built-in Messages

When the value of an ADF Faces component fails validation, or cannot be converted by a converter, the component displays the resulting `FacesMessage` instance.

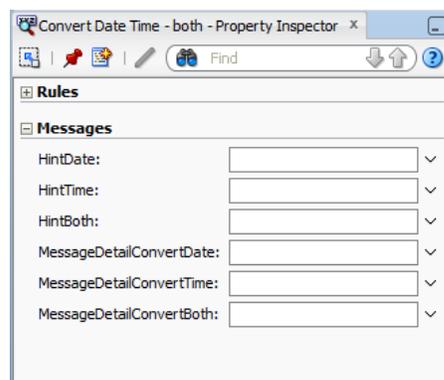
For example, entering a date that does not match the `dateStyle` attribute of the converter results in an error message, as shown in [Figure 19-19](#).

Figure 19-19 Validation Error at Runtime



You can override the default validator and converter hint and error messages for either a component instance, or globally for all instances. To define a custom message for a component instance you set attributes to the detail messages to be displayed. The actual attributes vary according to the validator or converter. [Figure 19-20](#) shows the attributes that you can populate to override the messages for the `convertDateTime` converter, as displayed in the Properties window.

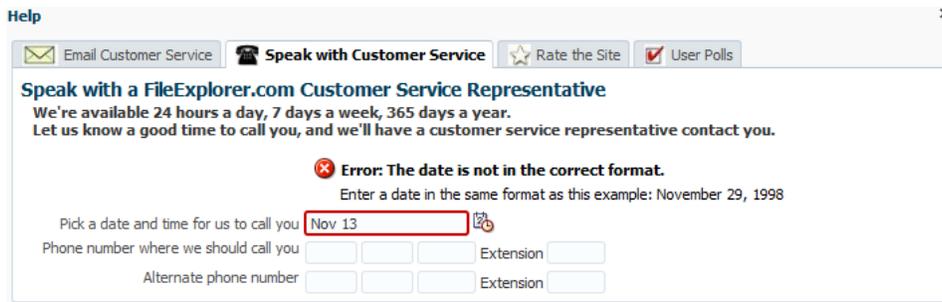
Figure 19-20 Message Attributes on a Converter



To define an error message that will be used by all instances of the component, you need to create an entry in a resource bundle that will override the default message.

If you do not want messages to be displayed in the note window, you can use the `message` component, and messages will be displayed inline with the component. [Figure 19-21](#) shows how messages are displayed using the `message` component.

Figure 19-21 Use the message Component to Display Messages Inline



JSF pages in an ADF Faces application use the `document` tag, which among other things, handles displaying all global messages (those not associated with a component) in a popup window. However, if you want to display global messages on the page instead, use the `messages` component.

 **Note:**

To format the message using HTML tags, you must enclose the message within `<html></html>` tags. For example:

```
<html><b>error</b> message details</html>
```

The following HTML tags are allowed in error messages:

- ``
- ``
- `<a>`
- `<i>`
- ``
- `
`
- `<hr>`
- ``
- ``
- ``
- `<p>`
- `<tt>`
- `<big>`
- `<small>`
- `<pre>`

How to Define Custom Validator and Converter Messages for a Component Instance

To override the default validator and converter messages for a single component instance, set values for the different message attributes.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Hints and Error Messages for Validation and Conversion](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Message Components](#).

To define a validator or converter message:

1. In the Structure window, select the converter or validator for which you want to create the error message.
2. In the Properties window, expand the **Messages** section and enter a value for the attribute for which you want to provide a message.

The values can include dynamic content by using parameter placeholders such as {0}, {1}, {2}, and so on. For example, the `messageDetailConvertDate` attribute on the `convertDateTime` converter uses the following parameters:

- {0} the label that identifies the component
- {1} the value entered by the user
- {2} an example of the format expected by the component.

Tips:

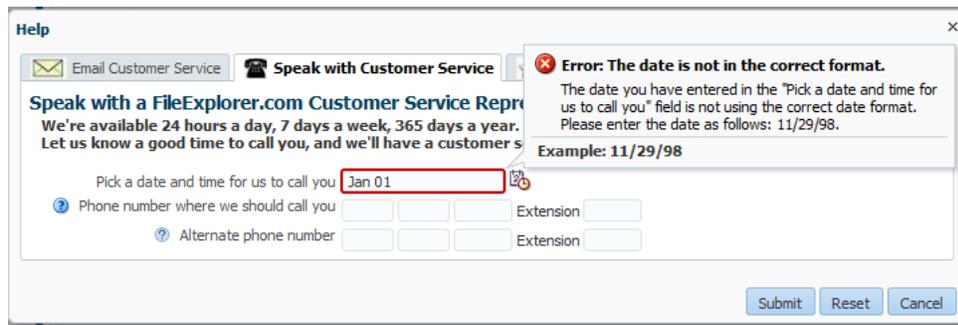
- If your application uses bidirectional or right-to-left display, do not start the message with the expected format parameter {2}, as it may not display correctly in Internet Explorer.
- You should try not to display the format parameter {1} in the message. Displaying the user input might be considered as a security violation.

Using these parameters, you can create the following message:

The date you have entered in the \"{0}\" field is not using the correct date format. Please enter the date as follows: {2}.

The error message would then be displayed as shown in [Figure 19-22](#). Note that you might need to change the value `Type` attribute of the converter to match the parameters.

Figure 19-22 Detail Message at Runtime



Tip:

Use the dropdown menu to view the property help, which includes the parameters accepted by the message.

If the text to be used is stored in a resource bundle, use the dropdown list to select **Select Text Resource**. Use the Select Text Resource dialog to either search for appropriate text in an existing bundle, or to create a new entry in an existing bundle. For information about using resource bundles, see [Internationalizing and Localizing Pages](#).

Note:

The message text is for the detail message of the `FacesMessage` object. If you want to override the summary (the text shown at the top of the message), you can only do this globally. See [How to Define Custom Validator and Converter Messages for All Instances of a Component](#).

How to Define Custom Validator and Converter Messages for All Instances of a Component

Instead of changing the message string per component instance with the `messageDetail[XYZ]` attributes, you can override the string globally so that the custom string will be displayed for all instances. The global messages are handled by key/value pairs in a message bundle. You can override summary, detail, and hint messages.

To globally override a default validator or converter message:

1. Refer to [Message Keys for Converter and Validator Messages](#) to determine the message key for the message you want to override. For example, to override the detail message displayed when the input value exceeds the maximum value length, you would use the key `org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM_detail`, as shown in [af:validateLength](#).

2. Create or open a message bundle. For procedures how to create message bundles, see [How to Create a Resource Bundle as a Property File or an XLIFF File](#).

3. Add the key override to the message bundle. For example, to override the message for the input value length, you might add:

```
org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM_detail: Your value exceeds the limit.
```

4. If the message bundle is a new resource bundle, you need to register the bundle with the application using the `faces-config.xml` file, following the procedures in [How to Register a Resource Bundle in Your Application](#). However, use the `<message-bundle>` tag, rather than the `<resource-bundle>` tag.

How to Display Component Messages Inline

Instead of having a component display its messages in the note window, use the `message` component to display the messages inline on the page. In order for the `message` component to display the correct messages, associate it with a specific component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Hints and Error Messages for Validation and Conversion](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Message Components](#).

To display component messages inline:

1. In the Structure window, select the component that will display its messages using the `message` component. If not already set, enter an ID for the component.
2. In the Components window, from the Text and Selection panel, drag a **Message** and drop it where you want the message to be displayed on the page.
3. Use the dropdown menu for the `for` attribute to select **Edit**.
4. In the Edit Property dialog, locate the component for which the `message` component will display messages. Only components that have their ID set are valid selections.

Note:

The message icon and message content that will be displayed are based on what was given when the `FacesMessage` object was created. Setting the `messageType` or `message` attributes on the `message` component causes the `messageType` or `message` attribute values to be displayed at runtime, regardless of whether or not an error has occurred. Only populate these attributes if you want the content to always be displayed when the page is rendered.

How to Display Global Messages Inline

Instead of displaying global messages in a popup window for the page, display them inline using the `messages` component.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Hints and Error Messages for Validation and Conversion](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Message Components](#).

To display global messages inline:

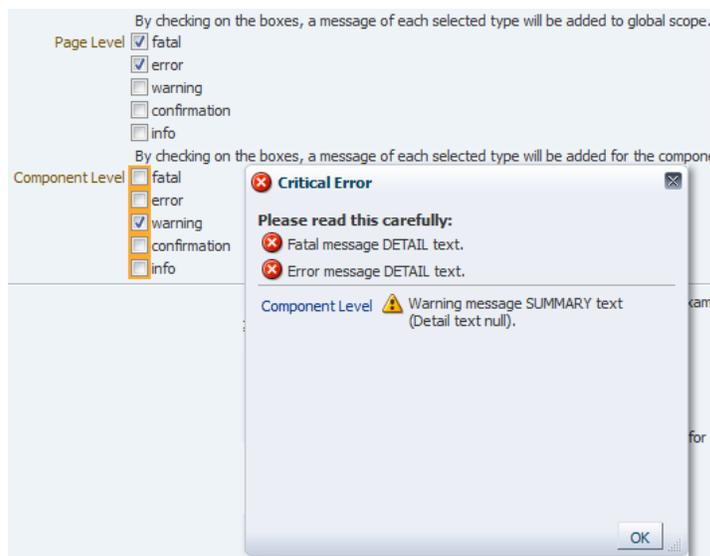
1. In the Components window, from the Text and Selection panel, drag a **Messages** and drop it onto the page where you want the messages to be displayed.
2. In the Properties window, set the following attributes:
 - `globalOnly`: By default, ADF Faces displays global messages (messages that are not associated with components) followed by individual component messages. If you want to display only global messages in the box, set this attribute to `true`. Component messages will continue to be displayed with the associated component.
 - `inline`: Set to `true` to show messages at the top of the page. Otherwise, messages will be displayed in a dialog.

What Happens at Runtime: How Messages Are Displayed

ADF Faces uses the standard JSF messaging API. JSF supports a built-in framework for messaging by allowing `FacesMessage` instances to be added to the `FacesContext` object using the `addMessage(java.lang.String clientId, FacesMessage message)` method. Component-level messages are associated with a specific component based on any client ID that was passed to the `addMessage` method, and global-level messages, which are not associated with a component because no client ID was passed to the `addMessage` method.

When conversion or validation fails on an `EditableValueHolder` ADF Faces component, `FacesMessages` objects are automatically added to the message queue on the `FacesContext` instance, passing in that component's ID. These messages are then displayed in the note window for the component. ADF Faces components are able to display their own messages. You do not need to add any tags.

Similarly, the `document` tag handles and displays all global `FacesMessages` objects (those that do not contain an associated component ID), as well as component `FacesMessages`. Like component messages, you do not need to add any tags for messages to be displayed. Whenever a global message is created (or more than one component message), all messages in the queue will be displayed in a popup window, as shown in [Figure 19-23](#).

Figure 19-23 Global and Component Messages Displayed by the Document**Tip:**

While ADF Faces provides messages for validation and conversion, you can add your own `FacesMessages` objects to the queue using the standard JSF messaging API. When you do so, ADF Faces will display icons with the message based on the message level, as follows:

Fatal	
Error	
Warning	
Confirmation	
Info	

Grouping Components with a Single Label and Message

The ADF Faces `panelLabelAndMessage` component allows you to group components and use a single label. You can group any components using this attribute and use a single label to the group.

By default, ADF Faces input and select components have built-in support for label and message display. If you want to group components and use a single label, wrap the components using the `panelLabelAndMessage` component.

For example, the File Explorer application collects telephone numbers using four separate `inputText` components; one for the area code, one for the exchange, one for the last four digits, and one for the extension. Because a single label is needed, the four `inputText` components are wrapped in a `panelLabelAndMessage` component, and

the label value is set on that component. However, the input component for the extension requires an additional label, so an `outputText` component is used. The following example shows the JSF code for the `panelLabelAndMessage` component.

```
<af:panelLabelAndMessage labelAndAccessKey="#{explorerBundle['help.telephone']}"
    helpTopicId="HELP_TELEPHONE_NUMBER"
    labelStyle="vertical-align: top;
    padding-top: 0.2em;">
  <af:inputText autoTab="true" simple="true" maximumLength="3"
    columns="3">
    <af:convertNumber type="number" integerOnly="true"/>
  </af:inputText>
  <af:inputText autoTab="true" simple="true" maximumLength="3"
    columns="3">
    <af:convertNumber type="number" integerOnly="true"/>
  </af:inputText>
  <af:inputText autoTab="true" simple="true" maximumLength="4"
    columns="4">
    <af:convertNumber type="number" integerOnly="true"/>
  </af:inputText>
  <af:outputText value="#{explorerBundle['help.extension']}/>
  <af:inputText simple="true" columns="4">
    <af:convertNumber type="number" integerOnly="true"/>
  </af:inputText>
</af:panelLabelAndMessage>
```

Figure 19-24 shows how the `panelLabelAndMessage` and nested components are displayed in a browser.

Figure 19-24 Examples Using the `panelLabelAndMessage` Component

Phone number where we should call you Extension

The `panelLabelAndMessage` component also includes an `End` facet that can be used to display additional components at the end of the group. Figure 19-25 shows how the telephone number fields would be displayed if the `End` facet was populated with an `outputText` component.

Figure 19-25 End Facet in a `panelLabelAndMessage` Component

Phone number where we should call you Extension End facet text

Use a `panelGroupLayout` component within a `panelLabelAndMessage` component to group the components for the required layout. For information about using the `panelGroupLayout` component, see [Grouping Related Items](#).

You set the `simple` attribute to `true` on each of the input components so that their individual labels are not displayed. However, you may want to set a value for the `label` attribute on each of the components for messaging purposes and for accessibility.

 **Tip:**

If you have to use multiple `panelLabelAndMessage` components one after another, wrap them inside an `af:panelFormLayout` component, so that the labels line up properly. For information about using the `panelFormLayout` component, see [Arranging Content in Forms](#).

Group and wrap components using the `panelLabelAndMessage` component. The `panelLabelAndMessage` component can be used to wrap any components, not just those that typically display messages and labels.

How to Group Components with a Single Label and Message

You use the `panelLabelAndMessage` component to group components and display a single label for that group.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Grouping Components with a Single Label and Message](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Message Components](#).

To arrange form input components with one label and message:

1. Add input or select components as needed to the page.

For each input and select component:

- Set the `simple` attribute to `true`.
- For accessibility reasons, set the `label` attribute to a label for the component.

2. In the Structure window, select the input or select components created in Step 1. Right-click the selection and choose **Surround With > Panel Label And Message**.

3. With the `panelLabelAndMessage` component selected, in the Properties window, set the following:

- **label:** Enter the label text to be displayed for the group of components.
- **for:** Use the dropdown menu to choose Edit. In the Edit Property dialog, select the ID of the child input component. If there is more than one input component, select the first component.

Set the `for` attribute to the first `inputComponent` to meet accessibility requirements.

If one or more of the nested input components is a required component and you want a marker to be displayed indicating this, set the `showRequired` attribute to `true`.

4. To place content in the `End` facet, drag and drop the desired component into the facet.

Because facets on a JSP or JSPX accept one child component only, if you want to add more than one child component, you must wrap the child components inside a

container, such as a `panelGroupLayout` or `group` component. Facets on a Facelets page can accept more than one component.

 **Tip:**

If the facet is not visible in the visual editor:

- a. Right-click the `panelLabelAndMessage` component in the Structure window.
- b. From the context menu, choose **Facets - Panel Label And Message >facet name**. Facets in use on the page are indicated by a checkmark in front of the facet name.

Displaying Help for Components

ADF Faces allows you to display help messages in three formats: instruction help, definition help, and external URL help. The help text is displayed from an external source.

ADF Faces provides a framework that allows you to create and display three different types of help whose content comes from an external source, rather than as text configured on the component. Because it is not configured directly on the component, the content can be used by more than one component, saving time in creating pages and also allowing you to change the content in one place rather than everywhere the content appears.

You can display help messages in three formats; instruction help, definition help, and external URL help.

Instruction Help

Instruction help displays help text in a note window when the focus is on the component. [Figure 19-26](#) shows the help message in a note window.

Figure 19-26 Instruction Text for a Component



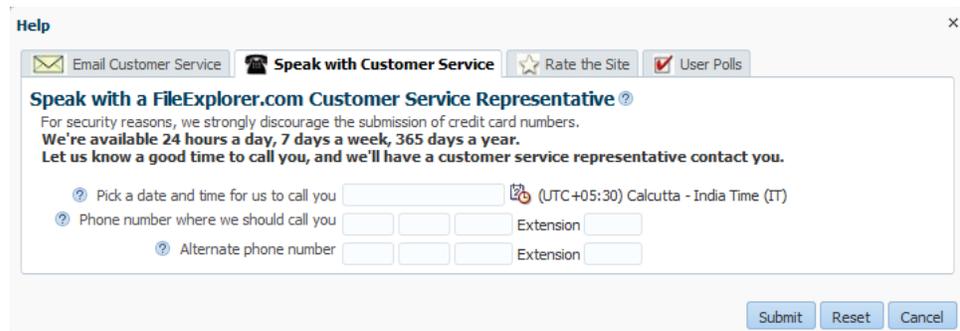
Usually, you use the instruction help to show instructions about how to use the component. You can also use HTML tags to format the help message, but not all HTML tags are supported (see tag documentation of `af:outputFormatted` component for supported HTML tags). [Figure 19-27](#) shows the HTML formatted help message in a note window.

Figure 19-27 HTML Formatted Instruction Help



Some components display instruction help as a description within the component. Where instruction help is displayed depends on the component with which it is associated. For example, the `panelHeader` and Search panel components display instruction help within the header, as shown in [Figure 19-28](#).

Figure 19-28 Instruction Text for panelHeader



[Table 19-1](#) shows the components that support Instruction help.

Table 19-1 Components That Support Instruction Help

Supported Components	Help Placement	Example
Input components, Choose Color, Choose Date, Quick Query	Note window, on focus only	
Select components	Note window, on hover and focus	

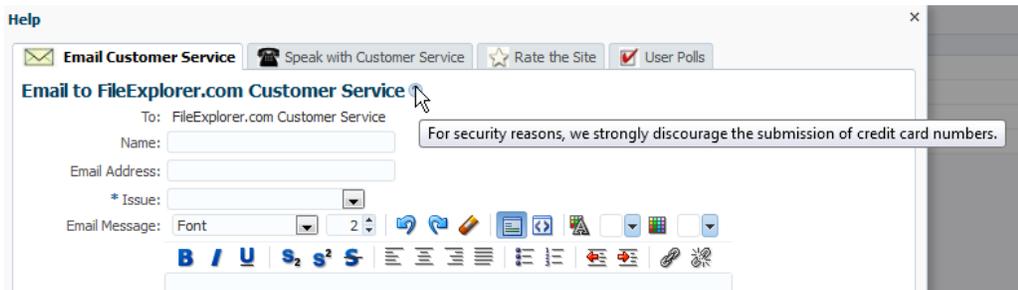
Table 19-1 (Cont.) Components That Support Instruction Help

Supported Components	Help Placement	Example
Panel Header, Panel Box, Query	Text below header text	

Definition Help

Definition help is like a standard tip where the content appears in a message box. However, instead of appearing when the user hovers the cursor over the component, definition help provides a help icon (a blue circle with a question mark). Or you can use a skinning attribute instead display a dotted line under the text (see [What You May Need to Know About Skinning and Definition Help](#)). When the user hovers the cursor over the icon, the content is displayed as shown in [Figure 19-29](#).

Figure 19-29 Definition Text for a Component

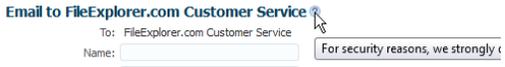
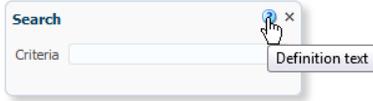
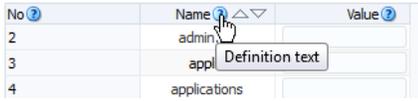


Note that the tip message does not support HTML formatting and the tip message must be less than 80 characters because some older versions of browsers do not support long messages. [Table 19-2](#) shows the components that support definition help.

Table 19-2 Components That Support Definition Help

Supported Components	Help Icon Placement	Example
All input components, Select components, Choose Color, Choose Date, Query components	Before the label, or if no label exists, at the start of the field	

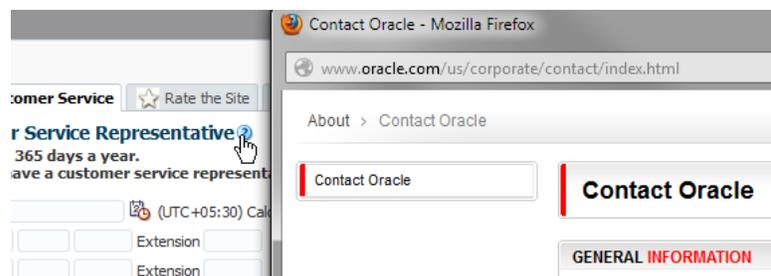
Table 19-2 (Cont.) Components That Support Definition Help

Supported Components	Help Icon Placement	Example
Panel Header, PanelBox, Show Detail Header	End of header text	
Panel Window, Dialog	Next to close icon in header	
Columns in table and tree	Below header text	

External URL Help

External URL help allows you to link to an existing web page for your help content. When the help icon is clicked, the web page opens in a separate browser window, as shown in [Figure 19-30](#). You can also use JavaScript to open the help window based on any client-based event.

Figure 19-30 External URL Help



For information about using JavaScript to open the external help, rather than the help icon, see [How to Use JavaScript to Launch an External Help Window](#).

Help Providers

Help providers store the text of the help (or a URL, in the case of external help) for an application, and are registered in the `META-INF/adf-settings.xml` file. ADF Faces supports the following help providers for instruction and definition help:

- The `ResourceBundleHelpProvider` help provider enables you to create resource bundles that hold the help content.
- The `ELHelpProvider` help provider enables you to create XLIFF files that get converted into maps.
- A managed bean that contains a map of help text strings.

External URL help uses a class that implements the `getExternalURL` method.

For information about creating help providers, see [How to Create Help Providers](#).

You can also use a combination of the different help providers, or create your own help provider class.

Creating Help

To create help messages, you do the following:

1. For instruction and definition help, determine which help provider type you want to use (either a resource bundle, an XLIFF file, or a managed bean), and then implement the required artifacts. These help providers will contain the actual help text. See [How to Create Help Providers](#).
2. For external help, you need to create the an external help provider that will contain the URLs to the external content. See [How to Create an External URL Help Provider](#).
3. Register the help providers, specifying the unique prefix that will be used to access the provider's help. See [How to Register the Help Provider](#).
4. Have the UI components access the help contained in the providers by using the component's `helpTopicId` attribute. A `helpTopicId` attribute contains the following:
 - The prefix that is used by the provider of the help
 - The topic nameSee [How to Access Help Content from a UI Component](#).
5. For external URL help, you can optionally have the help window launch in response to an event, instead of having the user click the help icon. See [How to Use JavaScript to Launch an External Help Window](#).

How to Create Help Providers

Procedures for creating help providers differ, depending on the type of help provider you want to create. For instruction and definition help, you can use resource bundles, an XLIFF file, or a managed bean. For external URL help, you create a Java class. You can also create a custom help provider.

How to Create a Resource Bundle-Based Provider

The `ResourceBundleHelpProvider` class provides a basic `HelpProvider` instance. You can store help text within standard resource bundle property files and use the `ResourceBundleHelpProvider` class to deliver the content.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To create resource bundle-based help:

1. Create a properties file that contains the topic ID and help text for each help topic. The topic ID must contain the following:
 - The prefix that will be used by this provider, for example, `BUNDLE`.
 - The topic name, for example, `PHONE_NUMBER`.
 - The help type, for example, `DEFINITION`.

For example, a topic ID might be `BUNDLE_PHONE_NUMBER_DEFINITION`.

If you are using multiple help providers, ensure that you use unique prefix for each help provider.

Note:

All prefixes under which help providers are registered must be unique. It is also not permissible for one prefix to begin with the same characters as another prefix. For example, if help providers have already been registered for the two prefixes `AAB` and `AC`, then the following prefixes are all invalid and will cause an exception to be thrown at registration time: `AABC`, `A`, `AA`, `AC`, `ACB`. However, the following are valid: `AAD`, `AB`, and so on.

UI components access the help content based on the topic name. Therefore, if you use the same topic name for two different types of help, then both types of help will be displayed by the UI component.

The following example shows an example resource bundle properties file with two topic IDs.

```
BUNDLE_CUST_SERVICE_EMAIL_DEFINITION=For security reasons, we strongly
discourage the submission of credit card numbers.
BUNDLE_PHONE_NUMBER_DEFINITION=We only support calling phone numbers in the
United States at this time.
```

2. Register the resource bundle as a help provider in the `adf-settings.xml` file and then configure the ADF Faces components to use the help. See [How to Register the Help Provider](#) and [How to Access Help Content from a UI Component](#).

How to Create an XLIFF Provider

You can store the help text in XLIFF XML files and use the `ELHelpProvider` class to deliver the content. This class translates the XLIFF file to a map of strings that will be used as the text in the help.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To create XLIFF help:

1. Create an XLIFF file that defines your help text, using the following elements within the `<body>` tag:
 - `<trans-unit>`: Enter the topic ID. This must contain the prefix, the topic name, and the help type, for example, `HELP_PHONE_NUMBER_DEFINITION`. In this example, `HELP` will become the prefix used to access the XLIFF file. `PHONE_NUMBER` is the topic name, and `DEFINITION` is the type of help. If you are using multiple help providers, ensure that you use unique prefix for each help provider.

Note:

All prefixes under which help providers are registered must be unique. It is also not permissible for one prefix to begin with the same characters as another prefix. For example, if help providers have already been registered for the two prefixes `AAB` and `AC`, then the following prefixes are all invalid and will cause an exception to be thrown at registration time: `AABC`, `A`, `AA`, `AC`, `ACB`. However, the following are valid: `AAD`, `AB`, and so on.

UI components access the help content based on the topic name. Therefore, if you use the same topic name for two different types of help, then both types of help will be displayed by the UI component.

- `<source>`: Create as a direct child of the `<trans-unit>` element and enter the help text.
- `<target>`: Create as a direct child of the `<trans-unit>` element and leave it blank. This element is used to hold translated help text.
- `<note>`: Create as a direct child of the `<trans-unit>` element and enter a description of the help text.

The following example shows an example of an XLIFF file that contains two topics.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.1" xmlns="urn:oasis:names:tc:xliff:document:1.1">
  <file source-language="en" original="this" datatype="xml">
    <body>
      <trans-unit id="HELP_PHONE_NUMBER_DEFINITION">
        <source>Phone Number Definition</source>
        <target/>
        <note>
          We only support calling phone numbers in the United States at this
```

```

time.
    </note>
</trans-unit>
<trans-unit id="HELP_CUST_SERVICE_EMAIL_INSTRUCTIONS">
  <source>Customer Service Email Instructions</source>
  <target/>
  <note>
    For security reasons, we strongly discourage the submission of credit
    card numbers.
  </note>
</trans-unit>
</body>
</file>
</xliff>

```

2. Register the XLIFF file as a help provider in the `adf-settings.xml` file and then configure the ADF Faces components to use the help. See [How to Register the Help Provider](#) and [How to Access Help Content from a UI Component](#).

How to Create a Managed Bean Provider

To implement a managed bean provider, create a managed bean that contains a map of strings that will be used as the text in the help. Managed bean help providers use the `ELHelpProvider` class to deliver the help.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To create managed bean help:

1. Create a managed bean that returns a map of strings, each of which is the ID and content for a help topic, as shown in the following example.

```

public class ELHelpProviderMapDemo
{
    public ELHelpProviderMapDemo()
    {
    }

    /* To use the ELHelpProvider, the EL expression must point to a Map, otherwise
    * you will get a coerceToType error. */

    public Map<String, String> getHelpMap()
    {
        return _HELP_MAP;
    }

    static private final Map<String, String> _HELP_MAP =
        new HashMap<String,
String>();
    static
    {
        _HELP_MAP.put("HELP_CUST_SERVICE_EMAIL_DEFINITION", "For security reasons,
        we strongly discourage the submission of credit card
numbers");
        _HELP_MAP.put("HELP_PHONE_NUMBER_DEFINITION", "We only support calling
phone
        numbers in the United States at this time");
    }
}

```

```
}
```

The first string must contain the prefix, the topic name, and the help type, for example, `HELP_CUST_SERVICE_EMAIL_DEFINITION`. In this example, `HELP` will become the prefix used to access the bean. `CUST_SERVICE_EMAIL` is the topic name, and `DEFINITION` is the type of help. The second string is the help text. If you are using multiple help providers, ensure that you use unique prefix for each help provider.

 **Note:**

All prefixes under which help providers are registered must be unique. It is also not permissible for one prefix to begin with the same characters as another prefix. For example, if help providers have already been registered for the two prefixes `AAB` and `AC`, then the following prefixes are all invalid and will cause an exception to be thrown at registration time: `AABC`, `A`, `AA`, `AC`, `ACB`. However, the following are valid: `AAD`, `AB`, and so on.

UI components access the help content based on the topic name. Therefore, if you use the same topic name for two different types of help, then both types of help will be displayed by the UI component.

 **Note:**

If you wish to use external URL help, create a subclass of the `ELHelpProvider` class. See [How to Create an External URL Help Provider](#).

2. Register the managed bean as a help provider in the `adf-settings.xml` file and then configure the ADF Faces components to use the help. See [How to Register the Help Provider](#) and [How to Access Help Content from a UI Component](#).

How to Create an External URL Help Provider

To use an external URL as help, you must implement the `getExternalURL` method.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To use External URL help:

1. Create a class that implements the `getExternalURL` method.

If you are using a resource bundle based help, ensure that the class that implements the `getExternalURL` method extends the `ResourceBundleHelpProvider` class.

If you are using XLIFF based or managed bean based help, ensure that the class that implements the `getExternalURL` method extends the `ELHelpProvider` class.

The following example shows the `DemoHelpProvider` class that extends the `ResourceBundleHelpProvider` class.

```
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.help.ResourceBundleHelpProvider;

public class DemoHelpProvider extends ResourceBundleHelpProvider
{
    public DemoHelpProvider()
    {
    }

    @Override
    protected String getExternalUrl(FacesContext context, UIComponent component,
                                    String topicId)
    {
        if (topicId == null)
            return null;

        if (topicId.contains("HELP_CONTACT_URL"))
        {
            return "http://www.oracle.com/us/corporate/contact/index.html";
        }
        else
            return null;

        if (topicId.contains("HELP_OTN_URL") )
        {
            return "http://www.oracle.com/technetwork/index.html";
        }
        else
            return null;
    }
}
```

The following example shows the `DemoELHelpProvider` class that extends the `ELHelpProvider` class.

```
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.help.ResourceBundleHelpProvider;

public class DemoELHelpProvider extends ELHelpProvider
{
    public DemoELHelpProvider()
    {
    }

    @Override
    protected String getExternalUrl(FacesContext context, UIComponent component,
                                    String topicId)
    {
        if (topicId == null)
            return null;

        if (topicId.contains("HELP_CONTACT_URL"))
        {
```

```
        return "http://www.oracle.com/us/corporate/contact/index.html";
    }
    else
        return null;

    if (topicId.contains("HELP_OTN_URL" )
    {
        return "http://www.oracle.com/technetwork/index.html";
    }
    else
        return null;
}
}
```

The preceding examples use the `HELP_CONTACT_URL` topic ID to open the Oracle Contact home page. To return different URLs, you would have to create separate if statements. They also use `HELP_OTN_URL` topic ID to open the Oracle Technology Network home page.

2. Register the class as a help provider in the `adf-settings.xml` file and then configure the ADF Faces components to use the help. See [How to Register the Help Provider](#) and [How to Access Help Content from a UI Component](#).

How to Create a Custom Java Class Help Provider

Instead of using one of the ADF Faces help providers, create your own. Create the actual text in some file that your help provider will be able to access and display. To create a Java class help provider, extend the `HelpProvider` class. For information about this class, refer to the *Java API Reference for Oracle ADF Faces*.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To create a Java class help provider:

1. Create a Java class that extends `oracle.adf.view.rich.help.HelpProvider`.
2. Create a public constructor with no parameters. You also must implement the logic to access and return help topics.

This class will be able to access properties and values that are set in the `adf-settings.xml` file when you register this provider. For example, all the ADF Faces providers use a property to define the actual source of the help strings.

3. To access a property in the `adf-settings.xml` file, create a method that sets a `String` property.

For example:

```
public void setMyCustomProperty(String arg)
```

4. To register the provider, from the `META-INF` node, open the `adf-settings.xml` file and add the following elements:
 - `<help-provider>`: Use the `prefix` attribute to define the prefix that UI components will use to access this help provider. This must be unique in the application.

 **Note:**

If the `prefix` attribute is missing, or is empty, then the help provider will be registered as a special default help provider. It will be used to produce help for help topic IDs that cannot be matched with any other help provider. Only one default help provider is permitted. All prefixes under which help providers are registered must be unique. It is also not permissible for one prefix to begin with the same characters as another prefix. For example, if help providers have already been registered for the two prefixes `AAB` and `AC`, then the following prefixes are all invalid and will cause an exception to be thrown at registration time: `AABC`, `A`, `AA`, `AC`, `ACB`. However, the following are valid: `AAD`, `AB`, and so on.

- `<help-provider-class>`: Create as a child element to the `<help-provider>` element and enter the fully qualified class path to the class created in Step 1.
- `<property>`: Create as a child element to the `<help-provider>` element and use it to define the property that will be used as the argument for the method created in Step 3.
- `<property-name>`: Create as a child element to the `<property>` element and enter the property name.
- `<value>`: Create as a child element to the `<property>` element and enter the value for the property.

How to Register the Help Provider

You register resource, XLIFF, and managed bean help providers, and an external URL help provider in the `adf-settings.xml` file.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To Register a Help Provider:

1. In the Applications window, from the `META-INF` node, open the `adf-settings.xml` file.
2. Click the Source tab, and add the following elements:
 - `<help-provider>`: Use the `prefix` attribute to define the prefix that UI components will use to access this help provider. This must be unique in the application.

 **Note:**

If the `prefix` attribute is missing, or is empty, then the help provider will be registered as a special default help provider. It will be used to produce help for help topic IDs that cannot be matched with any other help provider. Only one default help provider is permitted.

- `<help-provider-class>`: Create as a child element to the `<help-provider>` element.
 - If you are using a resource bundle, enter the value as `oracle.adf.view.rich.help.ResourceBundleHelpProvider`.
 - If you are using an XLIFF file, enter the value as `oracle.adf.view.rich.help.ELHelpProvider`.
 - If you are using a managed bean or external URL provider, enter the fully qualified class path.

 **Note:**

For external URL help, the provider class should reflect the class that your provider extended.

- `<property>`: Create as a child element to the `<help-provider>` element. The property defines the actual help source.
- `<property-name>`: Create as a child element to the `<property>` element, and enter a name for the source.
- `<value>`: Create as a child element to the `<property>` element, and enter the fully qualified class name of the helper. For example, the qualified class name of the resource bundle used in the ADF Faces Components Demo application is `oracle.adfdemo.view.resource.DemoResources`.
 - If you are using a resource bundle, enter the fully qualified class name of the resource bundle. For example, the qualified class name of the resource bundle used in the ADF Faces Components Demo application is `oracle.adfdemo.view.resource.DemoResources`.
 - If you are using an XLIFF file, enter an EL expression that resolves to the XLIFF file, wrapped in the `adfBundle` EL function, for example, `#{adfBundle['project1xliff.view.Project1XliffBundle']}`.
 - If you are using a managed bean, enter an EL expression that resolves to the help map on the managed bean.
 - If you are using external URL help, enter the fully qualified class name of the provider class you created.

The following example shows a resource bundle registered in the `adf-settings.xml` file.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="HELP_">
    <help-provider-class>
      oracle.adf.view.rich.help.ResourceBundleHelpProvider
    </help-provider-class>
    <property>
      <property-name>baseName</property-name>
      <value>oracle.adfdemo.view.resource.fileExplorer.helpStrings</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

The following example shows an XLIFF file registered in the `adf-settings.xml` file.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="HELP">
    <help-provider-class>
      oracle.adf.view.rich.help.ELHelpProvider
    </help-provider-class>
    <property>
      <property-name>helpSource</property-name>
      <value>#{adfBundle['project1xliff.view.Project1XliffBundle']}</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

The following example shows a bean registered in the `adf-settings.xml` file.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="HELP_">
    <help-provider-class>
      oracle.adf.view.rich.help.ELHelpProvider
    </help-provider-class>
    <property>
      <property-name>helpSource</property-name>
      <value>#{helpTranslationMap.helpMap}</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

The following example shows the external help provider registered in the `adf-settings.xml` file.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="HELP_">
    <help-provider-class>
      oracle.adf.view.rich.help.ResourceBundleHelpProvider
    </help-provider-class>
    <property>
      <property-name>baseName</property-name>
      <value>oracle.adfdemo.view.resource.DemoHelp</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

The following example shows an example of a custom help provider class registered in the `adf-settings.xml` file.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="MYAPP">
    <help-provider-class>
      oracle.adfdemo.view.webapp.MyHelpProvider
    </help-provider-class>
    <property>
      <property-name>myCustomProperty</property-name>
      <value>someValue</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

```
</property>
</help-provider>
</adf-faces-config>
</adf-settings>
```

How to Access Help Content from a UI Component

Use the `helpTopicId` attribute on components to access and display the help.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To access help from a component:

1. In the Structure window, select the component to which you want to add help. For a list of components that support help, see [Table 19-1](#) and [Table 19-2](#).
2. In the Properties window, expand the **Appearance** section, and enter a value for the `helpTopicId` attribute.

For definition and instruction help, this value should include the prefix to access the correct help provider and the topic name. It should not include the help type, as all help types registered with that name will be returned and displayed.

For example:

```
<af:inputText label="Customer Service E-Mail"
helpTopicId="HELP_CUST_SERVICE_EMAIL" />
```

This example will return both the definition and instruction help defined in the XLIFF file in the example in [How to Create an XLIFF Provider](#).

3. If you want to provide help for a component that does not support help, you can instead add an `outputText` component to display the help text, and then bind that component to the help provider, for example:

```
<af:outputFormatted
value="#{adfFacesContext.helpProvider['HELP_CUST_SERVICE_
EMAIL'].instructions}" />
```

This accesses the instruction help text.

How to Use JavaScript to Launch an External Help Window

If you are using an external URL help, by default, the user clicks a help icon to launch the help window. Instead, you can use JavaScript and a client event listener for a specific component's event to launch the help window.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. See [Displaying Help for Components](#).

To use JavaScript to launch an external help window:

1. Create a JavaScript function that uses the `launchHelp` API to launch a specific URL or page.

The following example shows the `launchHelp` function used to launch the `helpClient.jspx`.

```
<af:resource type="javascript">
  function launchHelp(event)
  {
    AdfPage.PAGE.launchHelpWindow("helpClient.jspx");
  }
</af:resource>
```

2. In the ADF Faces page, drag and drop the component whose client event will cause the `launchHelp` function to be called. You must set the `clientId` on this component to `true`.
3. In the Components window, from the Operations panel, drag and drop a **Client Listener** as a child to the component created in Step 2. Configure the `clientListener` to invoke the function created in Step 1. For information about using the `clientListener` tag, see [Listening for Client Events](#).

The following example shows the code used to assign the `click` event of button component to launch the `helpClient.jspx` page.

```
<af:toolbar id="tbl">
  <af:button text="Launch help window" id="ctbl"
    icon="/images/happy_computer.gif">
    <af:clientListener method="launchHelp" type="click"/>
  </af:button>
</af:toolbar>
<af:resource type="javascript">
  function launchHelp(event)
  {
    AdfPage.PAGE.launchHelpWindow("helpClient.jspx");
  }
</af:resource>
```

What You May Need to Know About Skinning and Definition Help

By default, when definition help is associated with a component, a question mark icon is displayed. The user clicks that icon to display the help. You can configure a custom skin to instead display a dotted line under the label of the component. This line denotes that the text is clickable. When the user clicks the text, the help is displayed.

Note:

If the component does not have label text, then only the icon can be displayed.

Figure 19-31 shows an `inputText` component with the default question mark icon.

Figure 19-31 Definition Help Icon

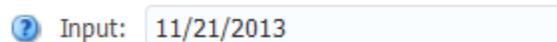


Figure 19-32 shows the same `inputText` component, but configured to display a link instead of the question mark.

Figure 19-32 Definition Help Text Link

Input:

You use the `-tr-help-def-location` skin selector key to configure the display. Set it to `icon` to display the icon. Set it to `label` to display the underline. The following example shows the help configured to display the underline.

```
af|help {  
  -tr-help-def-location: label;  
}
```

 **Note:**

External URL help also displays a link under the label text. If a component has both definition help and external help associated with it, then the icon will be displayed for the definition help.

Combining Different Message Types

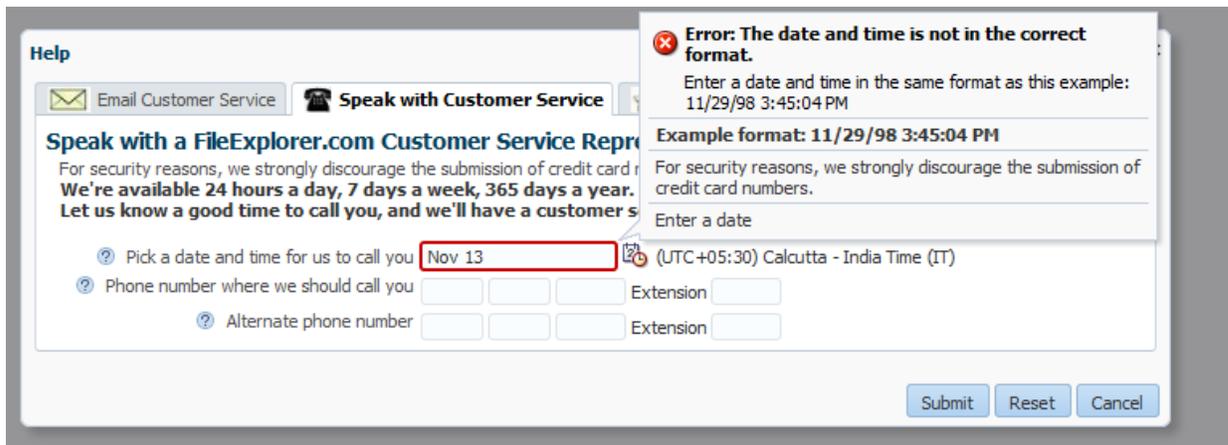
ADF Faces allows you to combine display messages for validation and conversion. You can use this to display various messages to users in a single window at one time; however, the messages are displayed in the default order of priority, such as error messages, hints, Instruction help and so on.

When you add help messages to input components that may already display messages for validation and conversion, ADF Faces displays the messages in the following order within the note window:

1. Validation and conversion error messages.
2. Validation and conversion hints.
3. For input and select components only, Instruction help. For `panelHeader` components, Instruction help is always displayed below the header.
4. Value for `shortDesc` attribute.

Figure 19-33 shows an `inputDate` component that contains a converter, instruction help, and a tip message.

Figure 19-33 Different Message Types Can Be Displayed at One Time



Working with Navigation Components

This chapter describes how to use ADF Faces navigation components to provide navigation in web user interfaces. This includes descriptions of how to use buttons and links to navigate and invoke functionality in addition to how to create page hierarchies. The chapter also describes how to use `train` components to navigate a multistep process.

This chapter includes the following sections:

- [About Navigation Components](#)
- [Common Functionality in Navigation Components](#)
- [Using Buttons and Links for Navigation](#)
- [Configuring a Browser's Context Menu for Links](#)
- [Using Buttons or Links to Invoke Functionality](#)
- [Using Navigation Items for a Page Hierarchy](#)
- [Using a Menu Model to Create a Page Hierarchy](#)
- [Creating a Simple Navigational Hierarchy](#)
- [Using Train Components to Create Navigation Items for a Multistep Process](#)

About Navigation Components

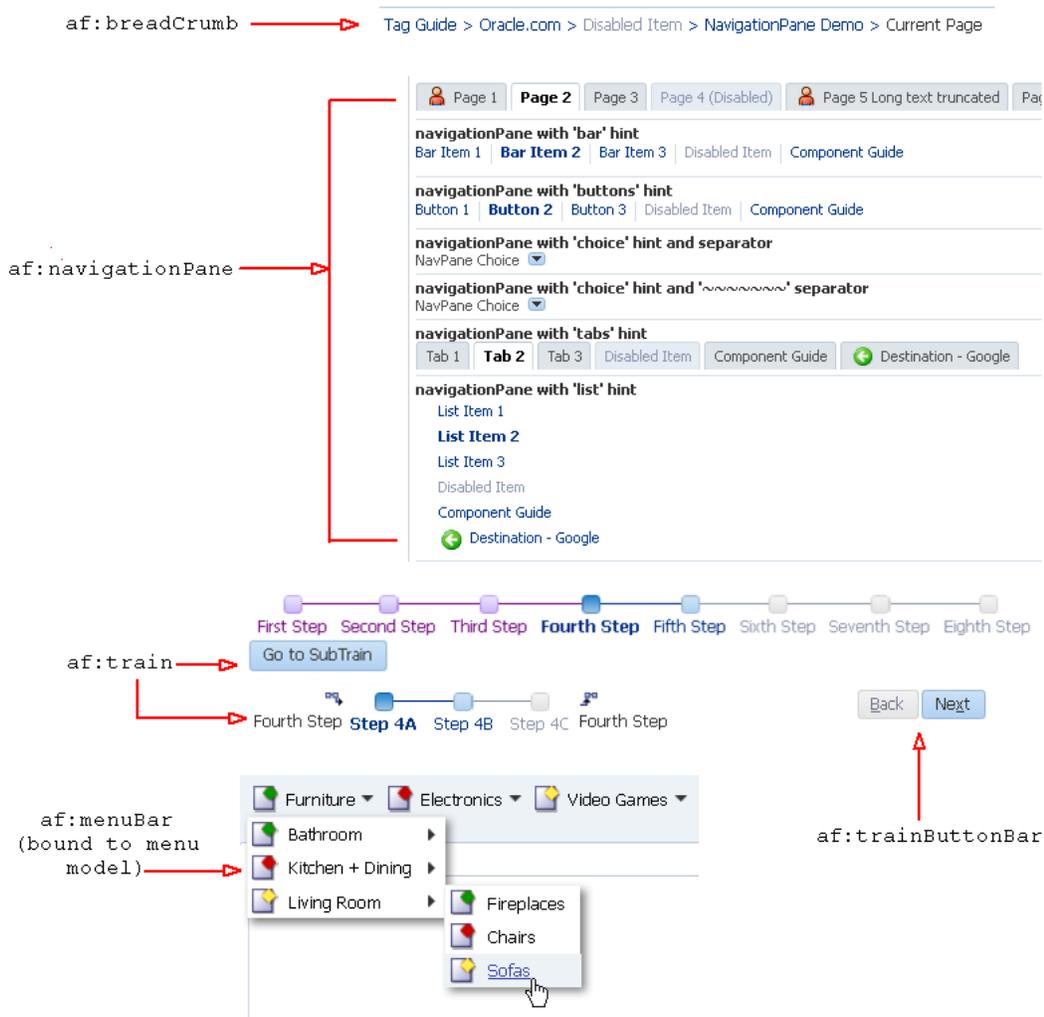
Using ADF Faces navigation components you can allow users to navigate another page or window and perform actions on data. Some of the navigation components are `af:breadcrumb`, `af:navigationPane`, `af:train`, and `af:menubar`.

Navigation components allow users to drill down for more information, to navigate to related pages or windows, and to perform specific actions on data and navigate at the same time. The common forms of navigation components are buttons and links, most of which can be used on their own and a few that can only be used in conjunction with other components.

Some components render navigable items such as tabs and breadcrumbs for navigating hierarchical pages and keeping track of the user's current location in the page hierarchy. Two components render links and buttons that you use specifically to guide users through a multistep task. You can also use the `button` or `link` components to fire partial page requests, and to implement popup dialogs and secondary windows (in conjunction with other ADF Faces tags and components). Navigation components can provide navigation with or without server-side actions.

[Figure 20-1](#) shows the different ADF Faces components that are used to provide navigation.

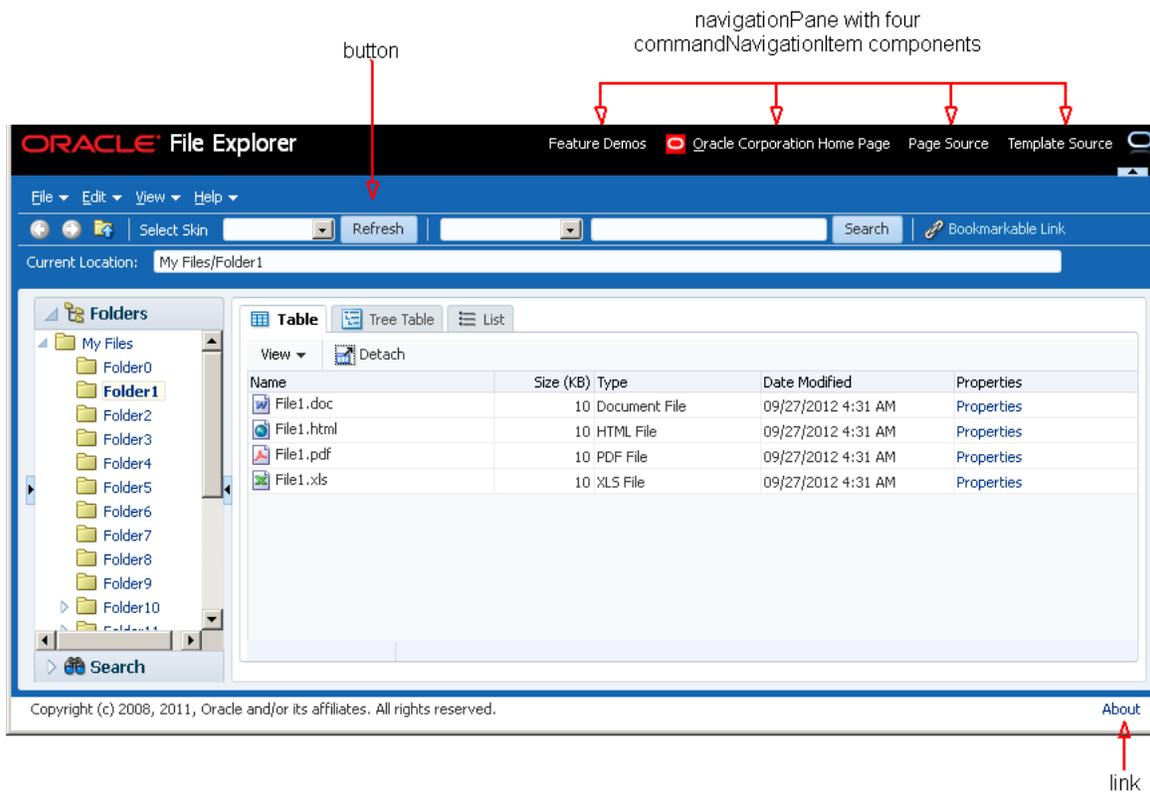
Figure 20-1 ADF Faces Navigation Components



Navigation Components Use Cases and Examples

Typical uses of navigation components are to create buttons and links to allow users to navigate to another page or window, to perform actions on data, or to perform actions and navigate at the same time. For example, as shown in Figure 20-2, the main page of the File Explorer application contains a `button` component that you click to refresh the page after making a skin selection and a `link` component that opens a popup window when clicked.

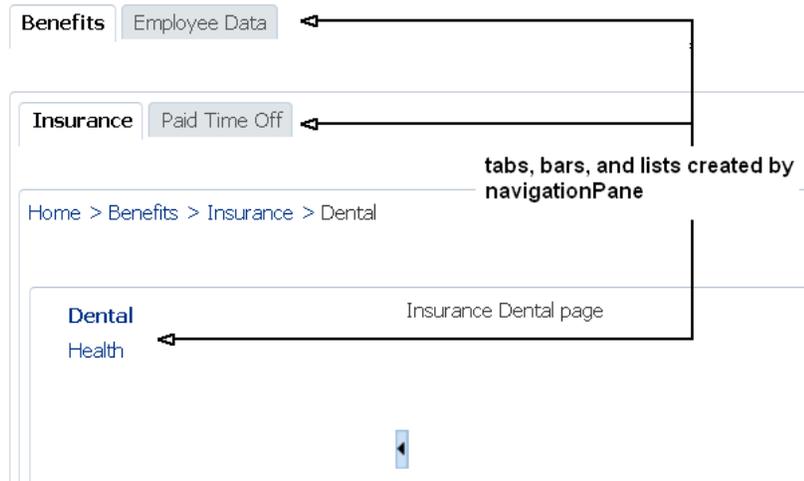
Figure 20-2 File Explorer Application Main Page



At the top right corner of the File Explorer application, there are four global application links. While you can use `link` components to provide the destinations for navigation, the File Explorer application uses the `navigationPane` and child `commandNavigationItem` components to provide links that either navigate directly to another location or deliver an action that results in navigation.

The `navigationPane` component also lets you organize application content in a meaningful structure and provides a navigation method for users to move through different content areas in the application to perform various functions. For example, a simple HR application might have pages that let employees check on company benefits, and pages for administration to view and create employee data, as shown in [Figure 20-3](#). The `navigationPane` component provides the structure with tabs, bars, or lists for example, and the child `commandNavigationItem` components provide the navigation links.

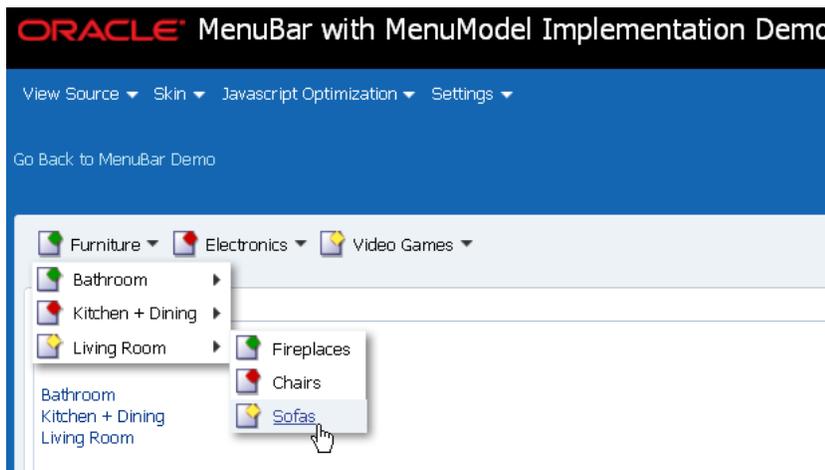
Figure 20-3 Page Showing Navigation Tab, Bar and List Links



The `navigationPane` component can also be used with a menu model, where the component is bound to the menu model managed bean. For complex page hierarchies, using a menu model is more efficient as the framework generates the correct number of navigation items in the structure on each page and also keeps track of which items are to be displayed as "selected".

The `menuBar` component can also be bound to a menu model to implement menus and submenus for navigating different levels in a page hierarchy. Most shopping websites use a system of menus to categorize shopping areas and provide a one-click action to a specific subcategory or item in the hierarchy. As shown in [Figure 20-4](#), the menu bar shows the first level of menu items at a glance. As the mouse cursor hovers over a menu, a submenu of more items display for the user to browse and choose. Typically you would not implement more than three levels of menu items.

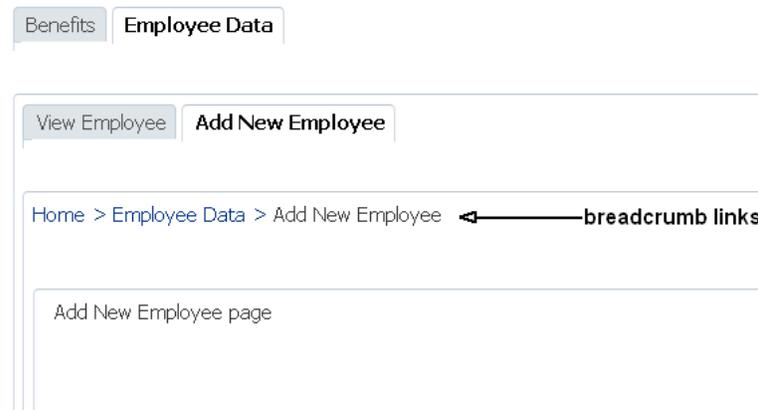
Figure 20-4 Page With Three Menus on a Bar with a Submenu Expanded



Whether you use `navigationPane` or `menuBar` (bound to a menu model) to create your page hierarchy, you can use the `breadcrumbs` component and a series of child

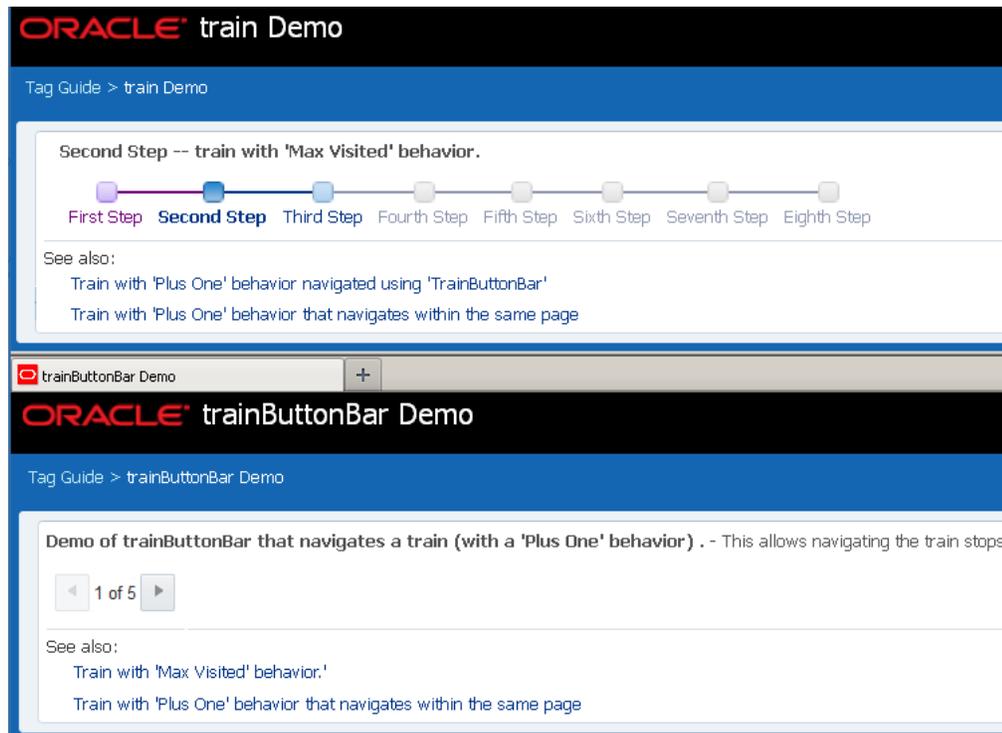
`commandNavigationItem` components to provide users with a visual indication to their current location in the page hierarchy. As shown in [Figure 20-5](#), the `breadcrumbs` component displays a line of text links starting from the root page down to the current page, which is always the last link. If you create your page hierarchy using a menu model, you can also bind the `breadcrumbs` component to the same menu model managed bean and let the framework dynamically generate the links for you.

Figure 20-5 Page Showing Horizontal Breadcrumb Links



The `train` component allows users to quickly see where they are in a multistep process and also navigate through that process. The `trainButtonBar` component provides additional navigation for a train process in the form of back and next buttons as arrows, and also gives information about the current and total number of pages as shown in [Working with Navigation Components](#).

Figure 20-6 ADF Faces Train and TrainButtonBar Demonstration Pages



Additional Functionality for Navigation Components

You may find it helpful to understand other ADF Faces features before you implement your navigation components. Additionally, once you have added these components to your page, you may find that you need to add functionality such as accessibility and localization. Following are links to other functionality that navigation components can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want the text displayed in a component to contain parameters that will resolve at runtime. See [How to Use the EL Format Tags](#).
- **Events:** Components fire both server-side and client-side events that you can have your application react to by executing some logic. See [Handling Events](#).
- **Partial page rendering:** ADF Faces navigation components can be used to trigger partial rerendering of components on a page. See [Rerendering Partial Page Content](#).
- **Accessibility:** You can make your navigation components accessible. See [Developing Accessible ADF Faces Pages](#).
- **Localization:** Instead of directly entering text for labels, you can use property files. These files allow you to manage translation of the text strings. See [Internationalizing and Localizing Pages](#).
- **Skins:** You can change the look and feel of navigation components by changing the skin. See [Customizing the Appearance Using Styles and Skins](#).

- **Touch Devices:** ADF Faces components may behave and display differently on touch devices. See [Creating Web Applications for Touch Devices Using ADF Faces](#).
- **Drag and Drop:** You can configure your components so that the user can drag and drop them to another area on the page. See [Adding Drag and Drop Functionality](#).

Common Functionality in Navigation Components

ADF Faces allows you to define rules for navigation by adding JSF navigation rules and cases in the configuration resource file (`faces-config.xml`) of the application. You must generate an `ActionEvent` event when users activate the component and using the `NavigationHandler` and `ActionListener` mechanisms you must select the navigation rules.

Like any JSF application, an application that uses ADF Faces components contains a set of rules for choosing the next page to display when a button or link (used on its own or within another navigation component) is clicked. You define the rules by adding JSF navigation rules and cases in the application's configuration resource file (`faces-config.xml`).

JSF uses an outcome string to select the navigation rule to use to perform a page navigation. ADF Faces navigation components that implement `javax.faces.component.ActionSource` interface generate an `ActionEvent` event when users activate the component. The JSF `NavigationHandler` and default `ActionListener` mechanisms use the outcome string on the activated component to find a match in the set of navigation rules. When JSF locates a match, the corresponding page is selected, and the Render Response phase renders the selected page. For information about the JSF lifecycle, see [Using the JSF Lifecycle with ADF Faces](#). Also note that navigation in an ADF Faces application may use partial page rendering. See [Rerendering Partial Page Content](#).

Using Buttons and Links for Navigation

ADF Faces provides various components for navigation, where buttons and links are the most common. Using these components you can allow users to navigate another location, submit requests, or to fire `ActionEvent` events. These components can also render images with optional text.

ADF Faces provides `button` and `link` components that can be used for navigation. Depending on your use case, you can configure these components to navigate directly to another location, to submit requests, and fire `ActionEvent` events.

Apart from the `button` and `link` components, ADF Faces also provides specialized components (`goMenuItem` and `commandMenuItem`) for use inside menus. See [Using Menus, Toolbars, and Toolboxes](#).

The `button` and `link` components can render images, along with optional text, as shown in [Figure 20-7](#) and [Figure 20-8](#). You can determine the position of the image relative to the optional text by setting a value for the `iconPosition` attribute. In addition, you can set different icons for when the user hovers over an icon, or the icon is depressed or disabled. You specify the image to use by setting a value for the `icon` attribute. The `button` and `link` components expand to the number of pixels required to accommodate the image that you specify to render within the component. If you do not

specify an image, the component renders using the minimum dimensions specified for the component in the web application's skin.

Figure 20-7 shows a number of the options that you can configure for the `button` component.

Figure 20-7 ADF Faces button Component

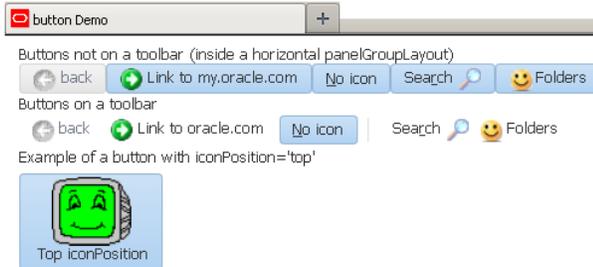
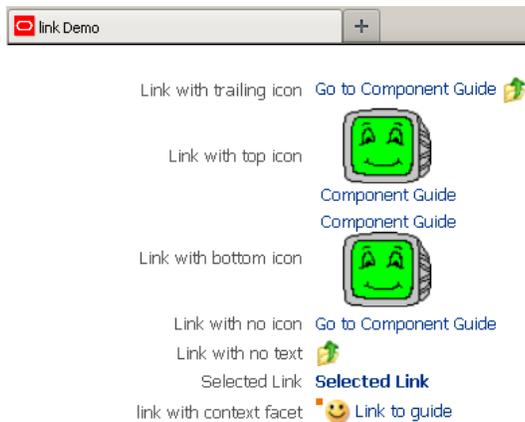


Figure 20-8 shows a number of the options that you can configure for the `link` component.

Figure 20-8 ADF Faces link Component



Using the ADF Faces `toolbar` component, you can provide additional functionality, such as a `popup` facet that opens popup menus from a `button` component. See [Using Toolbars](#).

The behavior of `button` and `link` components differ when you output your page in simplified mode for printing or email. The `link` component appears in print and email modes although it cannot be invoked while the `button` component does not render when you output a page in simplified mode for printing or email. For information about email and print output modes, see [Using Different Output Modes](#).

You can configure your application to allow end users to invoke a browser's context menu when they right-click an action component that renders a link. End users who right-click the link rendered by an action component may use a browser's context

menu to invoke an action that you do not want them to invoke (for example, open the link in a new window). See [Configuring a Browser's Context Menu for Links](#).

You can show a warning message to end users if the page that they attempt to navigate away from contains uncommitted data. Add the `checkUncommittedDataBehavior` component as a child to action components that have their `immediate` attribute set to `true`. If the user chooses not to navigate, the client event will be cancelled. You can add the `checkUncommittedDataBehavior` component as a child to the `af:button` and `af:link` components. For the warning message to appear to end users, the page must contain uncommitted data and you must have also set the document tag's `uncommittedDataWarning` attribute to `on`, as described in [How to Configure the document Tag](#).

 **Note:**

A warning message may also appear for uncommitted data if you set the document tag's `uncommittedDataWarning` tag to `on` and your page renders an ADF Controller bounded task flow that is configured as `critical`. See [How to Enable Implicit Save Points in Developing Fusion Web Applications with Oracle Application Development Framework](#).

How to Use Buttons and Links for Navigation and Deliver ActionEvents

Typically, you use action components like `button` and `link` to perform page navigation and to execute any server-side processing.

Before you begin:

It may help to understand how action component's attributes affect functionality. See [Using Buttons and Links for Navigation](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create and use action components:

1. In the Components window, from the General Controls panel, drag and drop the action component that you want to use onto the JSF page. More specifically, drag and drop a:
 - **Button** to create a `button` component.
 - **Link** to create a `link` component.
2. In the Properties window, expand the **Common** section and set the following:
 - **Text:** Specify the text to display.

 **Tip:**

Alternatively, you can use the `textAndAccessKey` attribute to provide a single value that defines the label along with the access key to use for the button or link. For information about how to define access keys, see [How to Define Access Keys for an ADF Faces Component](#).

- **Icon:** Set to the URI of the image file if you want to render an icon inside the component. If you render an icon, you can also set values for `hoverIcon`, `disabledIcon`, `depressedIcon`, and `iconPosition` in the **Appearance** section.

 **Tip:**

You can use either the `text` attribute (or `textAndAccessKey` attribute) or the `icon` attribute, or both.

- **IconPosition:** If you specified an icon, you can determine the position of the icon relative to the text by selecting a value from the dropdown list:
 - **<default> (leading):** Renders the icon before the text.
 - **trailing:** Renders the icon after the text.
 - **top:** Renders the icon above the text.
 - **bottom:** Renders the icon below the text.
- **Selected:** Set to `true` so that the component appears as selected when the page renders.
- **Action:** Set to an outcome string or to a method expression that refers to a backing bean action method that returns a logical outcome `String`. For information about configuring the navigation between pages, see [Defining Page Flows](#).

The default JSF `ActionListener` mechanism uses the outcome string to select the appropriate JSF navigation rule, and tells the JSF `NavigationHandler` what page to use for the Render Response phase. For information about using managed bean methods to open dialogs, see [Using Popup Dialogs, Menus, and Windows](#). For information about outcome strings and navigation in JSF applications, see the Java EE tutorial at <http://docs.oracle.com/javaee/index.html>.

 **Tip:**

The `actionListener` attribute can also be used for navigation when bound to a handler that returns an outcome. Usually, you should use this attribute only to handle user interface logic and not navigation.

For example, in the File Explorer application, the **Search** button in Search panel does not navigate anywhere. Instead, it performs a search. It has the following value for its `actionListener` attribute:

```
actionListener="#{explorer.navigatorManager.searchNavigator.  
                searchForFileItem}"
```

This expression evaluates to a method that actually performs the search.

3. Expand the **Behavior** section and set the following:
 - **Disabled:** Select `true` from the dropdown list if you want to show the component as a noninteractive button or link.
 - **PartialSubmit:** Select `true` from the dropdown list to fire a partial page request each time the component is activated. See [Using Partial Triggers](#).
 - **Immediate:** Select `true` from the dropdown list if you want to skip the Process Validations and Update Model phases. The component's action listeners (if any), and the default JSF `ActionListener` handler are executed at the end of the Apply Request Values phase of the JSF lifecycle. See [Using the Immediate Attribute](#).
4. Optionally, if you set the `immediate` attribute to `true` as described in Step 3, you can add the `af:checkUncommittedDataBehavior` component as a child to the action component to display a warning message to the user if the page contains uncommitted data. Drag **Check Uncommitted Data Behavior** from the Behavior group in the Operations panel of the Components window and drop it as a child of the action component you added in Step 1.

 **Note:**

You must have also set the document tag's `uncommittedDataWarning` attribute to `on`, as described in [How to Configure the document Tag](#).

Buttons and links can also be used to open secondary windows through these attributes: `useWindow`, `windowHeight`, `windowWidth`, `launchListener`, and `returnListener`. For information about opening secondary windows, see [Using the ADF Faces Dialog Framework Instead of Bounded Task Flows in *Developing Fusion Web Applications with Oracle Application Development Framework*](#).

To use buttons and links to invoke popups without writing any JavaScript code, see [Declaratively Invoking a Popup](#).

How to Use Buttons and Links for Navigation Without Delivering ActionEvents

You can use the `button` and `link` components to perform direct page navigation, without delivering an `ActionEvent` event.

Before you begin:

It may help to understand how the `button` and `link` components' attributes affect functionality. See [Using Buttons and Links for Navigation](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create buttons and links that navigate without delivering an `ActionEvent`:

1. In the Components window, from the General Controls panel, drag and drop the component that you want to use onto the JSF page. More specifically, drag and drop a:
 - **Button** to create a `button` component.
 - **Link** to create a `link` component.
2. In the Properties window, expand the **Common** section and set the following:
 - **Text**: Specify the text to display.

 **Tip:**

Instead, you can use the `textAndAccessKey` attribute to provide a single value that defines the label and the access key to use for the button or link. For information about how to define access keys, see [How to Define Access Keys for an ADF Faces Component](#).

- **Icon**: Set to the URI of the image file if you want to render an icon inside the component. If you render an icon, you can also set values for `hoverIcon`, `disabledIcon`, `depressedIcon`, and `iconPosition` in the **Appearance** section.

 **Tip:**

You can use either the `text` attribute (or `textAndAccessKey` attribute) or the `icon` attribute, or both.

- **IconPosition**: If you specified an icon, you can determine the position of the icon relative to the text by selecting a value from the dropdown list:
 - **<default> (leading)**: Renders the icon before the text.
 - **trailing**: Renders the icon after the text.
 - **top**: Renders the icon above the text.
 - **bottom**: Renders the icon below the text.

- **Selected:** Set to `true` so that the component appears as selected when the page renders.
- **Destination:** Set to the URI of the page to navigate to.

For example, set to the following to navigate to Oracle's web site:

```
destination="http://www.oracle.com"
```

- **TargetFrame:** Specify where the new page should display by selecting a value from the dropdown list:
 - **_blank:** The link opens the document in a new window.
 - **_parent:** The link opens the document in the window of the parent. For example, if the link appeared in a dialog, the resulting page would render in the parent window.
 - **_self:** The link opens the document in the same page or region.
 - **_top:** The link opens the document in a full window, replacing the entire page.
3. Expand the **Behavior** section and select `true` from the **Disabled** dropdown list if you want to show the component as a noninteractive button or link.

What You May Need to Know About Using Partial Page Navigation

As described in [Using Partial Page Navigation](#), you can configure an ADF Faces application to have navigation triggered through a partial page rendering request. When partial page navigation is turned on, partial page navigation for `GET` requests is automatically supported on the following components:

- `af:button`
- `af:link`
- `af:goMenuItem` (used within `af:menu` and `af:menuBar`)
- `af:commandNavigationItem` (used within `af:navigationPane`)

The only requirement is that the `destination` attribute on a supported component contain a relative URL of the application context root and begin with `"/`, such as `"/faces/myPage.jsf`, where `faces` is the URL mapping to the application's servlet defined in `web.xml` and `myPage.jsf` is the page to navigate. Because partial page navigation makes use of the hash (`#`) portion of the URL, you cannot use the hash portion for navigation to anchors within a page.

If the `targetFrame` attribute on a supported component is set to open the link in a new window, the framework automatically reverts to full page navigation.

Configuring a Browser's Context Menu for Links

The ADF Faces action components enable you to allow users to invoke actions by rendering the links at runtime. You can enable the context menu for `af:link`, `af:commandMenuItem`, `af:commandNavigationItem`, and others to invoke different actions by the users.

The action components that render links at runtime allow your end users to invoke actions. In addition you can configure your application so that the ADF Faces framework allows the end user's browser to render a context menu for these action

components. By default, the ADF Faces framework disables this context menu. The ADF Faces framework disables this context menu when no value is set for the `destination` attribute because the context menu may present menu options that invoke a different action (for example, open a link in a new window) to that specified by the action component. The components for which you can configure this behavior include the following:

- `af:link`
- `af:commandMenuItem` (used within an `af:menuBar` component)
- `af:commandNavigationItem` if no value is specified for the `destination` attribute, the ADF Faces framework enables the browser context menu in the following scenarios:
 - For the two anchors that `af:commandNavigationItem` renders when inside an `af:train` component
 - When an `af:commandNavigationItem` renders inside an `af:breadcrumbs` component
 - When an `af:commandNavigationItem` renders inside an `af:navigationPane` component (any `hint--tabs`, `bar`, `buttons`, `choice`, `list`)
- `af:panelTabbed`: the tabs and overflow indicators
- `af:panelAccordion`: the disclosure link and overflow indicators

You cannot configure this behavior for components that specify a destination and do not invoke an action. For example, an `af:commandNavigationItem` component where you specify a value for the `destination` attribute and no value for the `action` attribute.

How to Configure a Browser's Context Menu for Command Links

Set the value of the `oracle.adf.view.rich.ACTION_LINK_BROWSER_CONTEXT_SUPPRESSION` context parameter in your application's `web.xml` file to `no`.

Before you begin:

It may help to understand what action components you can configure this functionality for. See [Configuring a Browser's Context Menu for Links](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To configure a browser's context menu for a command link:

1. In the Applications window, expand the **WEB-INF** node and double-click **web.xml**.
2. In the overview editor, click the **Application** navigation tab and then click the **Add** icon next to the **Context Initialization Parameters** table to add an entry for the `oracle.adf.view.rich.ACTION_LINK_BROWSER_CONTEXT_SUPPRESSION` parameter and set it to `no`.
3. Save and close the `web.xml` file.

What Happens When You Configure a Browser's Context Menu for Command Links

If you followed the procedure outlined in [How to Configure a Browser's Context Menu for Command Links](#), JDeveloper writes a value to the `web.xml` file, as shown in [Example 20-1](#).

For information about ADF Faces configuration options in your application's `web.xml` file, see [Configuration in web.xml](#).

At runtime, end users can invoke a browser's context menu by right-clicking on the links rendered by certain components, as described in [Configuring a Browser's Context Menu for Links](#).

Example 20-1 Context Parameter to Configure a Browser's Context Menu

```
<context-param>
  <param-name>oracle.adf.view.rich.ACTION_LINK_BROWSER_CONTEXT_SUPPRESSION</param-name>
  <param-value>no</param-value>
</context-param>
```

Using Buttons or Links to Invoke Functionality

ADF Faces provides listener tags, along with action components, to execute when an action event is fired by an user. Some of the tags are `exportCollectionActionListener`, `fileDownloadActionListener`, and so on. You can also reset the input components by using a reset button.

In addition to using action components for navigation, ADF Faces also includes listener tags that you can use with action components to have specific functionality execute when the action event fires. Listener tags included with ADF Faces include:

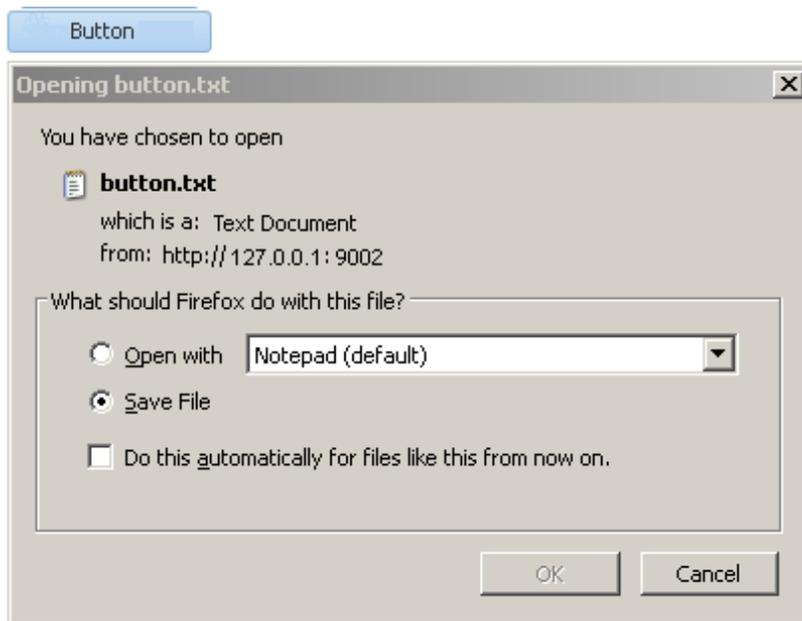
- `exportCollectionActionListener`: Use to export data from the `table`, `tree` and `treeTable` components to an Excel spreadsheet. See [Exporting Data from Table, Tree, or Tree Table](#).
- `fileDownloadActionListener`: Use to initiate a file download from the server to the local hard drive. See [How to Use an Action Component to Download Files](#).
- `resetListener`: Use to reset submitted values. However, no data model states will be altered. See [How to Use an Action Component to Reset Input Fields](#). If the input components render in a popup, see [Resetting Input Fields in a Popup](#).

If you want to reset the input components to their previous state, which was partially or fully submitted successfully to the server, then you can use a reset button. See [How to Add a Button to Reset the Form](#).

How to Use an Action Component to Download Files

You can create a way for users to download files by creating an action component such as a button and associating it with a `fileDownloadActionListener` tag. When the user selects or clicks the component, a popup dialog displays that allows the user to select different download options, as shown in [Figure 20-9](#).

Figure 20-9 File Download Dialog



Use the `fileDownloadActionListener` tag to allow an action component (for example, a button, link, or menu item) to send the contents of a file to an end user. You can also specify the content type or file name when you use this tag. Any value that you set for the action component's `partialSubmit` attribute is ignored at render time if you use the `fileDownloadActionListener` tag. The `fileDownloadActionListener` tag determines what type of submit the action component invokes based on the context. If you use the `fileDownloadActionListener` tag within a JSF portlet in your application, the action component invokes a partial submit (`partialSubmit="true"`). If you use the `fileDownloadActionListener` tag within an application that uses the ADF Faces servlet, the action component invokes a full submit (`partialSubmit="false"`).

 **Tip:**

For information about uploading a file to the server, see [Using File Upload](#).

After the content has been sent to the browser, how that content is displayed or saved depends on the option that the end user selects in the dialog. If the end user selects the **Open with** option, the application associated with that file type will be invoked to display the content. For example, a text file may result in the Notepad application being started. If the end user selects the **Save to Disk** option, depending on the browser, a popup dialog may appear to select a file name and a location in which to store the content.

The following example shows the tags of a button with the `fileDownloadActionListener` tag to download the file named `hello.txt` to the user.

```
<af:button value="Say Hello">
  <af:fileDownloadActionListener filename="hello.txt"
    contentType="text/plain; charset=utf-8"
```

```

        method="#{bean.sayHello}"/>
</af:button>

```

The following example shows a managed bean method that processes the file download.

```

public void sayHello(FacesContext context, OutputStream out) throws IOException{
    OutputStreamWriter w = new OutputStreamWriter(out, "UTF-8");
    w.write("Hi there!");
    . . .
}

```

If you use the `fileDownloadActionListener` tag from within a JSF portlet in your application, you can optionally add the parameters described in [Table 20-1](#) to the `web.xml` file of your application to configure the size and temporary location options for the file during download.

Table 20-1 Parameters to Add to `web.xml` File to Use `fileDownloadActionListener` in a Portlet

Parameter name	Data type	Description
<code>oracle.adf.view.rich.portal.FILE_DOWNLOAD_MAX_MEM</code>	Integer	Specify the maximum size in kilobytes of the file that the <code>fileDownloadActionListener</code> tag can store during a session. If the file exceeds the maximum size you specify, the application attempts to save the file to the hard drive in the location you specify for <code>FILE_DOWNLOAD_TEMP_DIR</code> . If you do not specify a value for this parameter in the <code>web.xml</code> file, it defaults to 100 kilobytes.
<code>oracle.adf.view.rich.portal.FILE_DOWNLOAD_MAX_DISK_SPACE</code>	Integer	Specify the maximum size in kilobytes of the file that the <code>fileDownloadActionListener</code> tag can download. If a file's size exceeds this value, an exception occurs and a log message is logged to the server's log file. If you do not specify a value for this parameter in the <code>web.xml</code> file, it defaults to 2000.
<code>oracle.adf.view.rich.portal.FILE_DOWNLOAD_TEMP_DIR</code>	String	Specify the temporary location where you store files during download. If you do not specify a value, it defaults to the directory specified by <code>java.io.tempDir</code> .

For information about configuring your `web.xml` file, see [Configuration in `web.xml`](#).

Before you begin:

It may help to understand how a component's attributes affect functionality. See [Using Buttons or Links to Invoke Functionality](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

You will need to complete this task:

Create an action component, as described in [Using Buttons and Links for Navigation](#).

To create a file download mechanism:

1. In the Components window, from the Operations panel, in the Listeners group, drag and drop the **File Download Action Listener** tag as a child to the action component.
2. In the Properties window set the following attributes:
 - **ContentType:** Specify the MIME type of the file, for example `text/plain`, `text/csv`, `application/pdf`, and so on.
 - **Filename:** Specify the proposed file name for the object. When the file name is specified, a Save File dialog will typically be displayed, though this is ultimately up to the browser. If the name is not specified, the content will typically be displayed inline in the browser, if possible.
 - **Method:** Specify the method that will download the file contents. The method takes two arguments, a `FacesContext` object and an `OutputStream` object. The `OutputStream` object will be automatically closed, so the sole responsibility of this method is to write all bytes to the `OutputStream` object.

For example, entries in the JSF page for a `button` component that uses the `fileDownloadActionListener` tag would be similar to the following:

```
<af:button text="Load File">
  <af:fileDownloadActionListener contentType="text/plain"
    filename="MyFile.txt"
    method="#{mybean.LoadMyFile}" />
</af:button>
```

How to Use an Action Component to Reset Input Fields

You can use the `resetListener` tag in conjunction with an action component to reset input values. When the end user invokes the action component, it resets all input values to null or empty. If you want to reset the input components to their previous state, which was partially or fully submitted successfully to the server, then you should use a reset button. See [How to Add a Button to Reset the Form](#).

If you use the `resetListener` tag to reset input components that render in a popup, you also need to set a value for the popup component's `resetEditableValues` property. For information about this use case, see [Resetting Input Fields in a Popup](#).

Before you begin:

It may help to understand how an action component's attributes affect functionality. See [Using Buttons or Links to Invoke Functionality](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

You will need to complete this task:

Create an action component, as described in [Using Buttons and Links for Navigation](#).

To use the reset listener tag:

1. In the Components window, from the Operations panel, in the Listeners group, drag a **Reset Listener** and drop it inside the action component that you created.
2. In the Insert Reset Listener dialog, specify the type of event that the `resetListener` tag activates in response to. For example, enter `action` so that the `resetListener` tag responds to an `actionEvent` returned by the action component's `actionListener` attribute.

Click **Help** in the Insert Reset Listener dialog to view a complete list of supported values.

Using Navigation Items for a Page Hierarchy

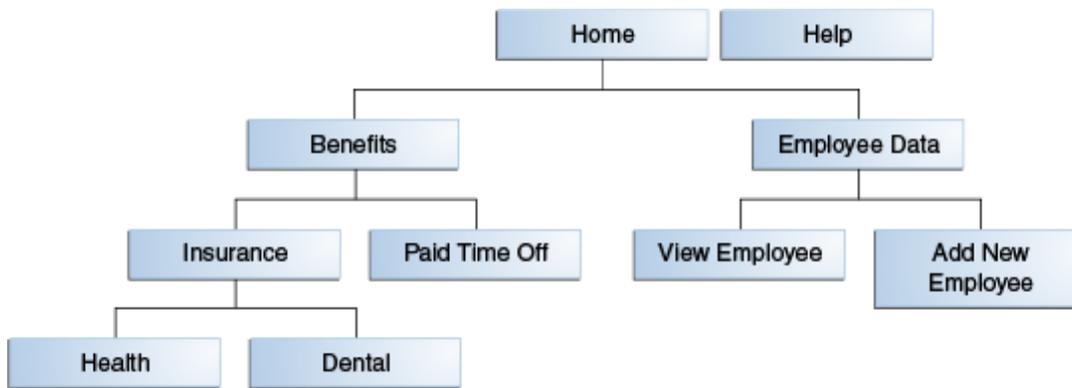
ADF Faces allows you to create page hierarchy by clicking a button or link. Users gain access to specific information by navigating through the links. You can provide either parent-child hierarchy or cross-link hierarchy for navigation.

Note:

If your application uses the Fusion technology stack with ADF Controller, then you should use ADF task flows and an `XMLMenuModel` implementation to create the navigation system for your application page hierarchy. See [Creating a Page Hierarchy Using Task Flows](#) in *Developing Fusion Web Applications with Oracle Application Development Framework*.

An application may consist of pages that are related and organized in a tree-like hierarchy, where users gain access to specific information on a page by drilling down a path of links. For example, [Figure 20-10](#) shows a simple page hierarchy with three levels of nodes under the top-level node, Home. The top-level node represents the root parent page; the first-level nodes, Benefits and Employee Data, represent parent pages that contain general information for second-level child nodes (such as Insurance and View Employee) that contain more specific information; the Insurance node is also a parent node, which contains general information for third-level child nodes, Health and Dental. Each node in a page hierarchy (except the root Home node) can be a parent and a child node at the same time, and each node in a page hierarchy corresponds to a page.

Figure 20-10 Benefits and Employee Page Hierarchy



Navigation in a page hierarchy follows the parent-child links. For example, to view Health information, the user would start drilling from the Benefits page, then move to the Insurance page where two choices are presented, one of which is Health. The path of selected links starting from Home and ending at Health is known as the focus path in the tree.

In addition to direct parent-child navigation, some cross-level or cross-parent navigation is also possible. For example, from the Dental page, users can jump to the Paid Time Off page on the second level, and to the Benefits page or the Employee Data page on the first level.

As shown in [Figure 20-10](#), the Help node, which is not linked to any other node in the hierarchy but is on the same level as the top-level Home node, is a global node. Global nodes represent global pages (such as a Help page) that can be accessed from any page in the hierarchy.

Typical widgets used in a web user interface for navigating a page hierarchy are tabs, bars, lists, and global links, all of which can be created by using the `navigationPane` component. [Figure 20-11](#) shows an example of how the hierarchy as illustrated in [Figure 20-10](#) could be rendered using the `navigationPane` and other components.

Figure 20-11 Rendered Benefits and Employee Data Pages



In general, tabs are used as first-level nodes, as shown in [Figure 20-11](#), where there are tabs for the Benefits and Employee Data pages. Second-level nodes, such as Insurance and Paid Time Off are usually rendered as bars, and third-level nodes, such as Health and Dental are usually rendered as lists. However, you may also use tabs for both first- and second-level nodes. Global links (which represent global nodes) are rendered as text links. In [Figure 20-11](#), the Home and Help global links are rendered as text links.

One `navigationPane` component corresponds to one level of nodes, whether they are first-, second-, or third-level nodes, or global nodes. Regardless of the type of items the `navigationPane` component is configured to render for a level, you always use the `commandNavigationItem` component to represent the items within the level.

The `navigationPane` component simply renders tabs, bars, lists, and global links for navigation. To achieve the positioning and visual styling of the page background, as shown in [Figure 20-16](#) and [Figure 20-17](#), you use the `decorativeBox` component as the parent to the first level `navigationPane` component. The `decorativeBox` component uses themes and skinning keys to control the borders and colors of its different facets. For example, if you use the default theme, the `decorativeBox` component body is white and the border is blue, and the top-left corner is rounded. If you use the medium theme, the body is a medium blue. The application must use the Skyros skin or a skin that extends from the Skyros skin. The Alta skin does not use themes. For information about using themes and skins, see [Customizing the Appearance Using Styles and Skins](#).

 **Tip:**

Because creating a page hierarchy requires that each page in the hierarchy use the same layout and look and feel, consider using a template to determine where the navigation components should be placed and how they should be styled. For more information, see [Using Page Templates](#).

On each page in simple hierarchies, you first use a series of `navigationPane` components to represent each level of the hierarchy. Then you add `commandNavigationItem` components as direct children of the `navigationPane` components for each of the links at each level. For example, to create the Health insurance page as shown in [Figure 20-11](#), you would first use a `navigationPane` component for each level displayed on the page, in this case it would be four: one for the global links, one for the first-level nodes, one for the second-level nodes, and one for the third-level nodes. You would then need to add `commandNavigationItem` components as children to each of the `navigationPane` components to represent the individual links (for example, you would add two `commandNavigationItem` child components to the third-level `navigationPane` component to represent the two third-level list items). If instead you were creating the Benefits page, as shown in [Figure 20-12](#), you would add only three `navigationPane` components (one each for the global, first, and second levels), and then add just the `commandNavigationItem` components for the links seen from this page.

Figure 20-12 First-Level Page



As you can see, with large page hierarchies, this process can be very time consuming and error prone. Instead of creating each of the separate `commandNavigationItem` components on each page, for larger hierarchies you can use an `XMLMenuModel` implementation and managed beans to dynamically generate the navigation items on the pages. The `XMLMenuModel` class, in conjunction with a metadata file, contains all the information for generating the appropriate number of hierarchical levels on each page, and the navigation items that belong to each level.

Then instead of using multiple `commandNavigationItem` components within each `navigationPane` component and marking the current items as selected on each page,

you declaratively bind each `navigationPane` component to the same `XMLMenuModel` implementation, and use one `commandNavigationItem` component in the `nodeStamp` facet to provide the navigation items. The `commandNavigationItem` component acts as a stamp for `navigationPane` component, stamping out navigation items for nodes (at every level) held in the `XMLMenuModel` object.

The `menuBar` component can also be used with the `XMLMenuModel` implementation to stamp out menu items for navigating a page hierarchy.

 **Note:**

If you want to create menus that can be used to cause some sort of change in an application (for example, a File menu that contains the commands Open and Delete), then see [Using Menus, Toolbars, and Toolboxes](#).

On any page, to show the user's current position in relation to the entire page hierarchy, you use the `breadcrumbs` component with a series of `commandNavigationItem` components or one `commandNavigationItem` component as a `nodeStamp`, to provide a path of links from the current page back to the root page (that is, the current nodes in the focus path).

For information about creating a navigational hierarchy using the `XMLMenuModel`, see [Using a Menu Model to Create a Page Hierarchy](#). For information about manually creating a navigational hierarchy, see [Creating a Simple Navigational Hierarchy](#).

How to Create Navigation Cases for a Page Hierarchy

Whether you use a menu model to create the navigation items for a page hierarchy or manually create the navigation items yourself, the JSF navigation model, through the default `ActionListener` mechanism, is used to choose the page to navigate to when users select a navigation item.

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Using Navigation Items for a Page Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create navigation cases for a page hierarchy:

1. In the Applications window, expand the **WEB-INF** node and double-click **faces-config.xml**.
2. Create one global JSF navigation rule that has the navigation cases for all the nodes in the page hierarchy.

For example, the page hierarchy shown in [Figure 20-10](#) has 10 nodes, including the global Help node. Thus, you would create 10 navigation cases within one global navigation rule in the `faces-config.xml` file, as shown in [Example 20-2](#).

For each navigation case, specify a unique outcome string, and the path to the JSF page that should be displayed when the navigation system returns an outcome value that matches the specified string.

For information about creating navigation cases in JDeveloper, see [Defining Page Flows](#).

Example 20-2 Global Navigation Rule for a Page Hierarchy in faces-config.xml

```
<navigation-rule>
  <navigation-case>
    <from-outcome>goHome</from-outcome>
    <to-view-id>/home.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHelp</from-outcome>
    <to-view-id>/globalhelp.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goEmp</from-outcome>
    <to-view-id>/empdata.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goBene</from-outcome>
    <to-view-id>/benefits.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goIns</from-outcome>
    <to-view-id>/insurance.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goPto</from-outcome>
    <to-view-id>/pto.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goView</from-outcome>
    <to-view-id>/viewdata.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goCreate</from-outcome>
    <to-view-id>/createemp.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHealth</from-outcome>
    <to-view-id>/health.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goDental</from-outcome>
    <to-view-id>/dental.jsf</to-view-id>
  </navigation-case>
</navigation-rule>
```

Using a Menu Model to Create a Page Hierarchy

ADF Faces provides a ready-to-use menu model when you want to create complex page hierarchies. It is a special kind of tree model with a collection of rows indexed by row keys.

 **Note:**

If your application uses the Fusion technology stack or ADF Controller, then you should use ADF task flows and an `XMLMenuModel` implementation to create the navigation system for your application page hierarchy. For details, see the "Creating a Page Hierarchy Using Task Flows" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

[Using Navigation Items for a Page Hierarchy](#) describes how you can create navigation items for a very simple page hierarchy using `navigationPane` components with multiple `commandNavigationItem` children components. Using the same method for more complex page hierarchies would be time consuming and error prone. It is inefficient and tedious to manually insert and configure individual `commandNavigationItem` components within `navigationPane` and `breadcrumbs` components on several JSF pages to create all the available items for enabling navigation. It is also difficult to maintain the proper selected status of each item, and to deduce and keep track of the breadcrumb links from the current page back to the root page.

For more complex page hierarchies (and even for simple page hierarchies), a more efficient method of creating a navigation system is to use a menu model. A **menu model** is a special kind of tree model. A **tree model** is a collection of rows indexed by row keys. In a tree, the current row can contain child rows (for information about a tree model, see [Displaying Data in Trees](#)). A menu model is a tree model that knows how to retrieve the `rowKey` of the node that has the current focus (the focus node). The menu model has no special knowledge of page navigation and places no requirements on the nodes that go into the tree.

The `XMLMenuModel` class creates a menu model from a navigation tree model. But `XMLMenuModel` class has additional methods that enable you to define the hierarchical tree of navigation in XML metadata. Instead of needing to create Java classes and configuring many managed beans to define and create the menu model (as you would if you used one of the other ADF Faces menu model classes), you create one or more `XMLMenuModel` metadata files that contain all the node information needed for the `XMLMenuModel` class to create the menu model.

 **Tip:**

Do not confuse the `navigationPane` component with the `panelTabbed` component. You use the `panelTabbed` component to display multiple tabbed content areas that can be hidden and displayed (see [Displaying or Hiding Contents in Panels](#)). However, the `panelTabbed` component cannot bind to any navigational model and the whole content must be available from within the page, so it has limited applicability.

To create a page hierarchy using a menu model, you do the following:

- Create the JSF navigation rule and navigation cases for the page hierarchy. See [How to Create Navigation Cases for a Page Hierarchy](#).

- Create the `XMLMenuModel` metadata. See [How to Create the Menu Model Metadata](#).
- Configure the managed bean for the `XMLMenuModel` class. The application uses the managed bean to build the hierarchy. This configuration is automatically done for you when you use the Create ADF Menu Model dialog in JDeveloper to create the `XMLMenuModel` metadata file. See [What Happens When You Use the Create ADF Menu Model Wizard](#).
- Create a JSF page for each of the hierarchical nodes (including any global nodes).

 **Tip:**

Typically, you would use a page template that contains a facet for each level of items (including global items and breadcrumbs) to create each JSF page. For example, the `navigationPane` component representing global items might be wrapped in a facet named `navigationGlobal`, and the `navigationPane` component representing first level tabs might be wrapped in a `navigation1` facet. For information about creating page templates, see [Creating and Reusing Fragments, Page Templates, and Components](#).

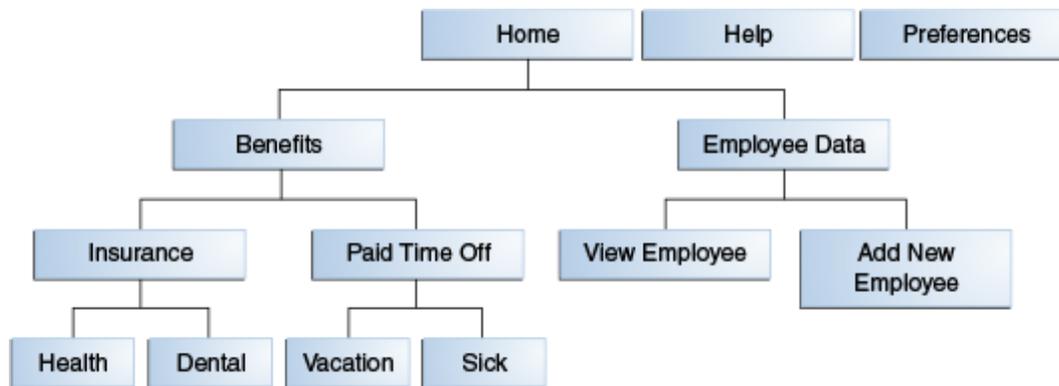
- On each page, bind the `navigationPane` and `breadcrumbs` components to the `XMLMenuModel` class. See [How to Bind the navigationPane Component to the Menu Model](#) and [How to Use the breadcrumbs Component with a Menu Model](#). To bind the `menuBar` component, see [How to Use the menuBar Component with a Menu Model](#).

How to Create the Menu Model Metadata

The `XMLMenuModel` metadata file is a representation of a navigation menu for a page hierarchy in XML format. You can use one or more `XMLMenuModel` metadata files to represent an entire page hierarchy. In an `XMLMenuModel` metadata file, the page hierarchy is described within the `menu` element, which is the root element of the file. Every `XMLMenuModel` metadata file is required to have a `menu` element and only one `menu` element is allowed in each file.

The other elements in the `XMLMenuModel` metadata file or hierarchy can be made up of item nodes, group nodes, and shared nodes. Item nodes represent navigable nodes (or pages) in the hierarchy. For example, say you wanted to build the hierarchy depicted in [Figure 20-13](#).

Figure 20-13 Sample Page Hierarchy



If you wanted each node in the hierarchy to have its own page to which a user can navigate, then in the metadata file you would create an item node for each page. You nest children nodes inside a parent node to create the hierarchy. However, say you did not need a page for the Employee Data node, but instead wanted the user to navigate directly to the View Employee page. You would then use a group node to represent the Employee Data page and use the group node's `idref` attribute to reference the page that opens (the View Employee page) when an end user clicks the Employee Data tab. The group node allows you to retain the hierarchy without needing to create pages for nodes that are simply aggregates for their children nodes.

The following example shows an `XMLMenuModel` metadata file that uses mostly item nodes and one group node to define the entire page hierarchy illustrated in [Figure 20-13](#).

```

<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <itemNode id="in01" focusViewId="/home.jsf" label="Home" action="goHome">
    <itemNode id="in1" focusViewId="/benefits.jsf" action="goBene"
      label="Benefits">
      <itemNode id="in11" focusViewId="/insurance.jsf" action="goIns"
        label="Insurance">
        <itemNode id="in111" focusViewId="/health.jsf" action="goHealth"
          label="Health"/>
        <itemNode id="in112" focusViewId="/dental.jsf" action="goDental"
          label="Dental"/>
        </itemNode>
        <itemNode id="in12" focusViewId="/pto.jsf" action="goPto"
          label="Paid Time Off">
          <itemNode id="in121" focusViewId="/vacation.jsf"
            action="goVacation" label="Vacation"/>
          <itemNode id="in122" focusViewId="/sick.jsf" action="goSick"
            label="Sick Pay"/>
          </itemNode>
        </itemNode>
      <groupNode id="gn2" idref="newEmp" label="Employee Data">
        <itemNode id="in21" focusViewId="/createemp.jsf" action="goCreate"
          label="Create New Employee"/>
        <itemNode id="in22" focusViewId="/viewdata.jsf" action="goView"
          label="View Data"/>
        </groupNode>
      </itemNode>
    <itemNode id="in02" focusViewId="/globalhelp.jsf" action="goHelp"
  
```

```

        label="Help"/>
    <itemNode id="in03" focusViewId="/preferences.jsf" action="goPref"
        label="Preferences"/>
</menu>

```

Within the root `menu` element, global nodes are any nodes that are direct children of the `menu` element. For example, the code in the previous example shows three global nodes, namely, Home, Help, and Preferences.

You can also nest menu models using shared nodes. Use this approach where you have sub trees in the hierarchy (for example, the Benefits tree) as it makes the page hierarchy easier to maintain. For example, you might create the entire Benefits tree as its own menu model metadata file (as shown in the following example) so that the menu model could be reused across an application.

```

<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
    <itemNode id="in1" focusViewId="/benefits.jsf" action="goBene"
        label="Benefits">
        <itemNode id="in11" focusViewId="/insurance.jsf" action="goIns"
            label="Insurance">
            <itemNode id="in111" focusViewId="/health.jsf" action="goHealth"
                label="Health"/>
            <itemNode id="in112" focusViewId="/dental.jsf" action="goDental"
                label="Dental"/>
        </itemNode>
        <itemNode id="in12" focusViewId="/pto.jsf" action="goPto"
            label="Paid Time Off">
            <itemNode id="in121" focusViewId="/vacation.jsf"
                action="goVacation" label="Vacation"/>
            <itemNode id="in122" focusViewId="/sick.jsf" action="goSick"
                label="Sick Pay"/>
        </itemNode>
    </itemNode>
</menu>

```

Once you have created the nodes as a separate menu model, then within the different hierarchies that need to use those nodes, you use a shared node to reference the Benefits menu model.

The following example shows an `XMLMenuModel` metadata file that uses item nodes, a shared node and a group node to define the same page hierarchy depicted in [Figure 20-13](#).

```

<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
    <itemNode id="in01" focusViewId="/home.jsf" label="Home" action="goHome">
        <sharedNode ref="#{benefits_menu}/>
        <groupNode id="gn2" idref="newEmp" label="Employee Data">
            <itemNode id="in21" focusViewId="/createemp.jsf" action="goCreate"
                label="Create New Employee"/>
            <itemNode id="in22" focusViewId="/viewdata.jsf" action="goView"
                label="View Data"/>
        </groupNode>
    </itemNode>
    <itemNode id="in02" focusViewId="/globalhelp.jsf" action="goHelp"
        label="Help"/>
    <itemNode id="in03" focusViewId="/preferences.jsf" action="goPref"
        label="Preferences"/>
</menu>

```

The `sharedNode` element references the managed bean that is configured for the Benefits `XMLMenuModel` metadata file. Whenever you use the Create ADF Menu Model wizard to create a metadata file, JDeveloper automatically adds the managed bean configuration for you.

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Using a Menu Model to Create a Page Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create the `XMLMenuModel` metadata:

 **Note:**

If your application uses ADF Controller, then this menu option will not be available to you. You need to instead use a bounded task flow to create the hierarchy. See *Creating a Page Hierarchy Using Task Flows* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

For information about the managed bean configuration that JDeveloper automatically adds for you in `faces-config.xml`, see [What Happens When You Use the Create ADF Menu Model Wizard](#).

For information about using resource bundles, see [Internationalizing and Localizing Pages](#).

1. In the Applications window, locate the project where you want to create the `XMLMenuModel` metadata file. Expand the **WEB-INF** node, right-click **faces-config.xml** and choose **Create ADF Menu Model**.
2. In the Create ADF Menu Model dialog, enter a file name for the `XMLMenuModel` metadata file, for example, `root_menu`.

 **Tip:**

If you are using more than one `XMLMenuModel` metadata file to define the page hierarchy, use the name `root_menu` only for the topmost (root) metadata file that contains references to the other submenu metadata files.

3. Enter a directory for the metadata file. By default, JDeveloper saves the `XMLMenuModel` metadata file in the **WEB-INF** node of the application.

When you click **OK**, JDeveloper displays a blank `XMLMenuModel` metadata file in the source editor, as shown in the following example.

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"></menu>
```

4. Select the **menu** node in the Structure window and enter the appropriate information in the Properties window.

Table 20-2 shows the attributes you can specify for the `menu` element.

Table 20-2 Menu Element Attributes

Attribute	Description
<code>resourceBundle</code>	Optional. This is the resource bundle to use for the labels (visible text) of the navigation items at runtime. For example, <code>org.apache.myfaces.demo.xmlmenuDemo.resource.MenuBundle</code> .
<code>var</code>	If using a resource bundle, specify an ID to use to reference the bundle in EL expressions for navigation item labels. For example, <code>#{bundle.somelabel}</code> . See the following example for a sample <code>XMLMenuModel</code> metadata file that uses a resource bundle.
<code>xmlns</code>	Required. Set to <code>http://myfaces.apache.org/trinidad/menu</code>

The following example shows sample `XMLMenuModel` metadata code that uses EL expressions to access a resource bundle for the navigation item labels.

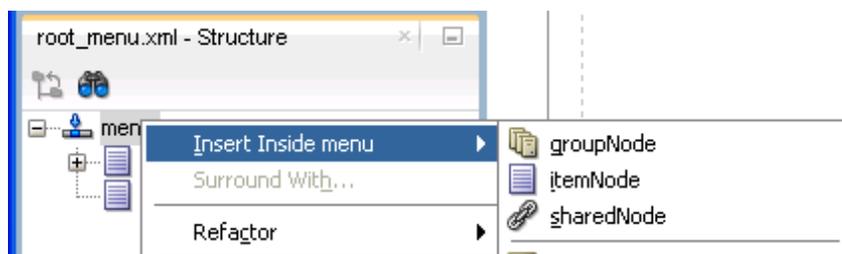
```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"
      resourceBundle="org.apache.myfaces.demo.xmlmenuDemo.resource.MenuBundle"
      var="bundle">
  <itemNode id="in1" label="#{bundle.somelabel1}" ../>
  <itemNode id="in2" label="#{bundle.somelabel2}" ../>
</menu>
```

 **Note:**

When you use a `sharedNode` element to create a submenu and you use resource bundles for the navigation item labels, it is possible that the shared menu model will use the same value for the `var` attribute on the root `menu` element. The `XMLMenuModel` class handles this possibility during parsing by ensuring that each resource bundle is assigned a unique hash key.

- In the Structure window, right-click **menu** and choose **Insert inside menu**, and then choose the desired element (`itemNode`, `groupNode`, or `sharedNode`) from the subsequent context menu, as shown in Figure 20-14, to add to the nodes in your hierarchy.

Figure 20-14 Context Menu for Inserting Elements into Menu



The elements can be one of the following:

- **itemNode**: Specifies a node that performs navigation upon user selection.
- **groupNode**: Groups child components; the `groupNode` itself does no navigation. Child nodes node can be `itemNode` or another `groupNode`.

For example, say you did not need a page for the Employee Data node, but instead, wanted the user to navigate directly to the View Employee page. You would then use a group node to represent the Employee Data page by specifying the `id` attribute of the desired child node as a value for the group node's `idref` attribute. The group node allows you to retain the hierarchy without needing to create pages for nodes that are simply aggregates for their children nodes.

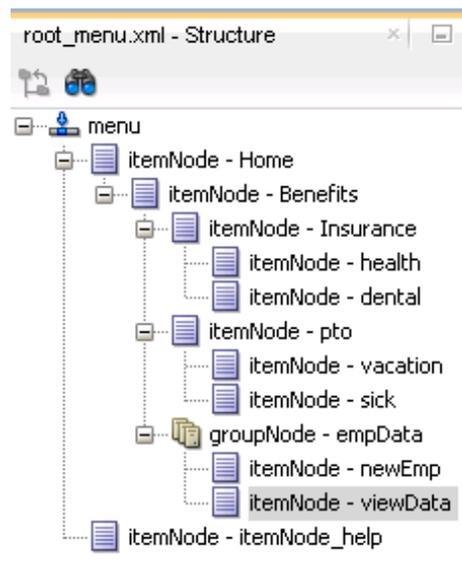
- **sharedNode**: References another `XMLMenuModel` instance. A `sharedNode` element is not a true node; it does not perform navigation nor does it render anything on its own.

You can insert a `sharedNode` element anywhere within the hierarchy. The code shown in the following example demonstrates how the `sharedNode` element adds a submenu on the same level as the first-level Employee Data node.

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"
  <itemNode id="in0" label="Home" ../>
    <sharedNode ref="#{shared_menu}"/>
    <itemNode id="in1" label="Employee Data" ../>
  </itemNode>
  <itemNode id="in01" label="Help" ../>
</menu>
```

As you build the `XMLMenuModel` metadata file, the tree structure you see in the Structure window exactly mirrors the indentation levels of the menu metadata, as shown in [Figure 20-15](#).

Figure 20-15 Tree Structure of `XMLMenuModel` Metadata in Structure Window



6. For each element used to create a node, set the properties in the Properties window, as described in [Table 20-3](#) for `itemNode` elements, [Table 20-4](#) for `groupNode` elements, and [Table 20-5](#) for `sharedNode` elements.

Table 20-3 itemNode Element Attributes

Attribute	Description
<code>action</code>	Specify either an outcome string or an EL method binding expression that returns an outcome string. In either case, the outcome string must match the <code>from-outcome</code> value to the navigation case for that node as configured in the <code>faces-config.xml</code> file.
<code>destination</code>	Specify the URI of the page to navigate to when the node is selected, for example, <code>http://www.oracle.com</code> . If the destination is a JSF page, the URI must begin with <code>"/faces"</code> . Alternatively, specify an EL method expression that evaluates to the URI. If both <code>action</code> and <code>destination</code> are specified, <code>destination</code> takes precedence over <code>action</code> .
<code>focusViewId</code>	Required. The URI of the page that matches the node's navigational result, that is, the <code>to-view-id</code> value of the navigation case for that node as specified in the <code>faces-config.xml</code> file. For example, if the action outcome of the node navigates to <code>/page_one.jsf</code> (as configured in the <code>faces-config.xml</code> file), then <code>focusViewId</code> must also be <code>/page_one.jsf</code> . The <code>focusViewId</code> does not perform navigation. Page navigation is the job of the <code>action</code> or <code>destination</code> attributes. The <code>focusViewId</code> , however, is required for the <code>XMLMenuModel</code> to determine the correct focus path.
<code>id</code>	Required. Specify a unique identifier for the node. As previous examples show, it is good practice to use "inX" for the ID of each <code>itemNode</code> , where for example, "inX" could be <code>in1</code> , <code>in11</code> , <code>in111</code> , <code>in2</code> , <code>in21</code> , <code>in211</code> , and so on.
<code>label</code>	Specify the label text to display for the node. Can be an EL expression to a string in a resource bundle, for example, <code>#{bundle.someLabel}</code> , where <code>bundle</code> must match the root menu element's <code>var</code> attribute value.
<code>any</code>	Specify any custom attribute for the <code>itemNode</code> .

A `groupNode` element does not have the `action` or `destination` attribute that performs navigation directly, but it points to a child node that has the `action` outcome or `destination` URI, either directly by pointing to an `itemNode` child (which has the `action` or `destination` attribute), or indirectly by pointing to a `groupNode` child that will then point to one of its child nodes, and so on until an `itemNode` element is reached. Navigation will then be determined from the `action` outcome or `destination` URI of that `itemNode` element.

Consider the `groupNode` code shown in the following example. At runtime, when users click `groupNode id="gn1"`, or `groupNode id="gn11"`, or `itemNode id="in1"`, the navigation outcome is `"goToSubTabOne"`, as specified by the first `itemNode`

reached (that is `itemNode id="id1"`). Table 20-4 shows the attributes you must specify when you use a `groupNode` element.

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns:"http://myfaces.apache.org/trinidad/menu">
  <groupNode id="gn1" idref="gn11" label="GLOBAL_TAB_0">
    <groupNode id="gn11" idref="in1" label="PRIMARY_TAB_0">
      <itemNode id="in1" label="LEVEL2_TAB_0" action="goToSubTabOne"
        focusViewId="/menuDemo/subtab1.jsf"/>
      <itemNode id="in2" label="LEVEL2_TAB_1" action="goToSubTabTwo"
        focusViewId="/menuDemo/subtab2.jsf"/>
    </groupNode>
    <itemNode id="in3" label="PRIMARY_TAB_1" focusViewId="/menuDemo/tab2.jsf"
      destination="/faces/menuDemo/tab2.jsf"/>
  </groupNode>
  <itemNode id="gin1" label="GLOBAL_TAB_1" action="goToGlobalOne"
    focusViewId="/menuDemo/global1.jsf"/>
  <itemNode id="gin2" label="GLOBAL_TAB_2"
    destination="/faces/menuDemo/global2.jsf"
    focusViewId="/menuDemo/global2.jsf"/>
</menu>
```

Table 20-4 GroupNode Element Attribute

Attribute	Description
<code>id</code>	A unique identifier for the group node. As shown in the previous example, it is good practice to use <code>gnX</code> for the ID of each <code>groupNode</code> , where for example, <code>gnX</code> could be <code>gn1</code> , <code>gn2</code> , and so on.
<code>idref</code>	Specify the ID of a child node, which can be an <code>itemNode</code> , or another <code>groupNode</code> . When adding a <code>groupNode</code> as a child node, that child in turn can reference another <code>groupNode</code> and so on, but eventually an <code>itemNode</code> child must be referenced as the last child. The <code>idref</code> attribute can contain more than one child ID, separated by spaces; the IDs are processed in the order they are listed.
<code>label</code>	Specify the label text to display for the group node. Can be an EL expression to a string in a resource bundle, for example, <code>#{bundle.somelabel}</code> .
<code>any</code>	Specify any custom attribute for the <code>groupNode</code> .

Table 20-5 sharedNode Element Attribute

Attribute	Description
<code>ref</code>	Specify the managed bean name of another <code>XMLMenuModel</code> class, as configured in the <code>faces-config.xml</code> file, for example, <code>#{shared_menu}</code> . At runtime, the referenced navigation menu is created, inserted as a submenu into the main (root) menu, and rendered.

What Happens When You Use the Create ADF Menu Model Wizard

When you use the Create ADF Menu Model wizard to create an `XMLMenuModel` metadata file, JDeveloper automatically configures for you a managed bean for the menu metadata file in the `faces-config.xml` file, using the metadata file name you provide as the managed bean name.

The following example shows part of the `faces-config.xml` file that contains the configuration of one `XMLMenuModel` metadata file. By default, JDeveloper uses the `oracle.adf.view.rich.model.MDSMenuModel` class as the managed bean class, and `request` as the managed bean scope, which is required and cannot be changed.

```
<managed-bean>
  <managed-bean-name>root_menu</managed-bean-name>
  <managed-bean-class>oracle.adf.view.
    rich.model.MDSMenuModel</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>createHiddenNodes</property-name>
    <value>>false</value>
  </managed-property>
  <managed-property>
    <property-name>source</property-name>
    <property-class>java.lang.String</property-class>
    <value>/WEB-INF/root_menu.xml</value>
  </managed-property>
</managed-bean>
```

In addition, the following managed properties are added by JDeveloper for the `XMLMenuModel` managed bean:

- `createHiddenNodes`: When `true`, specifies that the hierarchical nodes must be created even if the component's `rendered` attribute is `false`. The `createHiddenNodes` value is obtained and made available when the menu metadata source file is opened and parsed. This allows the entire hierarchy to be created, even when you do not want the actual component to be rendered.
- `source`: Specifies the menu metadata source file to use (for example, `/WEB-INF/root_menu.xml`).

Note:

The `createHiddenNodes` property must be placed before the `source` property, which JDeveloper does for you when the managed bean is automatically configured. The `XMLMenuModel` managed bean must have the `createHiddenNodes` value already set to properly parse and create the menu's XML metadata from the `source` managed property.

For each `XMLMenuModel` metadata file that you create in a project using the wizard, JDeveloper configures a managed bean for it in the `faces-config.xml` file. For example, if you use a `sharedNode` element in an `XMLMenuModel` to reference another `XMLMenuModel` metadata file, you would have created two metadata files. And JDeveloper would have added two managed bean configurations in the `faces-`

config.xml file, one for the main (root) menu model, and a second managed bean for the shared (referenced) menu model, as shown in the following example.

```
<!-- managed bean for referenced, shared menu model -->
<managed-bean>
  <managed-bean-name>shared_menu</managed-bean-name>
  <managed-bean-class>
    <managed-bean-class>oracle.adf.view.
      rich.model.MDSMenuModel</managed-bean-class>
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>createHiddenNodes</property-name>
    <value>>false</value>
  </managed-property>
  <managed-property>
    <property-name>source</property-name>
    <property-class>java.lang.String</property-class>
    <value>/WEB-INF/shared_menu.xml</value>
  </managed-property>
</managed-bean>
```

This means, if you use shared nodes in your XMLMenuModel metadata files, the faces-config.xml file will have a root menu model managed bean, plus menu model managed beans for any menu models referenced through shared nodes.

How to Bind the navigationPane Component to the Menu Model

Each node in the page hierarchy corresponds to one JSF page. On each page, you use one navigationPane component for each level of navigation items that you have defined in your XMLMenuModel metadata file, including global items. Levels are defined by a zero-based index number: Starting with global nodes in the metadata file (that is, direct children nodes under the menu element), the level attribute value is 0 (zero), followed by 1 for the next level (typically tabs), 2 for the next level after that (typically bars), and so on. For example, if you had a page hierarchy like the one shown in [Figure 20-13](#), you would use three navigationPane components on a page such as Home (for the three levels of navigation under the Home node), plus one more navigationPane component for the global nodes.

Tip:

Because the menu model dynamically determines the hierarchy (that is, the links that appear in each navigationPane component) and also sets the current nodes in the focus path as selected, you can practically reuse the same code for each page. You need to change only the page's document title, and add the specific page contents to display on that page.

Because of this similar code, you can create a single page fragment that has just the facets containing the navigationPane components, and include that fragment in each page, where you change the page's document title and add the page contents.

As described in [How to Create a Simple Page Hierarchy](#), you use the hint attribute to specify the type of navigation item you want to use for each hierarchical level (for

example, buttons, tabs, or bar). But instead of manually adding multiple `commandNavigationItem` components yourself to provide the navigation items, you bind each `navigationPane` component to the root `XMLMenuModel` managed bean, and insert only one `commandNavigationItem` component into the `nodeStamp` facet of each `navigationPane` component, as shown in the following example.

```
<af:navigationPane var="menuNode" value="#{root_menu}" level="0"
    hint="buttons">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
      action="#{menuNode.doAction}"
      icon="#{menuNode.icon}"
      destination="#{menuNode.destination}"
      visible="#{menuNode.visible}"
      rendered="#{menuNode.rendered}"/>
  </f:facet>
</af:navigationPane>
```

The `nodeStamp` facet and its single `commandNavigationItem` component, in conjunction with the `XMLMenuModel` managed bean, are responsible for:

- Stamping out the correct number of navigation items in a level.
- Displaying the correct label text and other properties as defined in the metadata. For example, the EL expression `#{menuNode.label}` retrieves the correct label text to use for a navigation item, and `#{menuNode.doAction}` evaluates to the action outcome defined for the same item.
- Marking the current items in the focus path as selected. You should not specify the `selected` attribute at all for the `commandNavigationItem` components.

 **Note:**

If there is no node information in the `XMLMenuModel` object for a particular hierarchical level (for example, level 3 lists), ADF Faces does not display those items on the page even though the page contains the `navigationPane` component code for that level.

If you want the navigation items to be styled, create a `decorativeBox` component by dragging and dropping a **Decorative Box** from the Layout panel of the Components window to the JSF page. Set the theme to determine how you want the tabs to appear. Valid values are:

- default
- light
- medium
- dark

Each value describes the look and feel applied to the application when you specify the theme value for the component. The application must use the Skyros skin or a skin that extends from the Skyros skin. The Alta skin does not use themes. You can change how the themes display. See [Customizing the Appearance Using Styles and Skins](#).

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Using a Menu Model to Create a Page Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To bind a `navigationPane` component to the menu model:

1. In the Components window, from the Layout panel, in the Interactive Containers and Headers group, drag and drop a **Navigation Pane** onto the JSF page for each level of the hierarchy.

For example, to create any of the pages as shown in the hierarchy in [Figure 20-13](#), you drag and drop four `navigationPane` components onto the JSF page.

2. For each `navigationPane` component, in the Properties window, expand the **Common** section and select one of the following types of navigation items from the **Hint** dropdown list to determine how the `navigationPane` displays:
 - **bar**: Displays the navigation items separated by a bar, for example the **Insurance** and **Paid Time Off** links in [Figure 20-17](#).
 - **buttons**: Displays the navigation items separated by a bar in a global area, for example the **Home** and **Help** links in [Figure 20-17](#).
 - **choice**: Displays the navigation items in a popup list when the associated dropdown icon is clicked. You must include a value for the `navigationPane` component's `icon` attribute and you can associate a label to the dropdown list using the `title` attribute.
 - **list**: Displays the navigation items in a bulleted list, for example the **Health** and **Dental** links in [Figure 20-17](#).
 - **tabs**: Displays the navigation items as tabs, for example the **Benefits** and **Employee Data** tabs in [Figure 20-17](#).

3. In the **Level** field, enter a number for the appropriate level of metadata in the `XMLMenuModel` metadata file. The `level` attribute is a zero-based index number: Starting with global nodes in the metadata file (that is, direct children nodes under the `menu` element), the `level` attribute value is 0 (zero), followed by 1 for the next level (typically tabs), 2 for the next level after that (typically bars), and so on.

The `commandNavigationItem` component is able to get its metadata from the metadata file through the `level` attribute on the parent `navigationPane` component. By default, if you do not specify a `level` attribute value, 0 (zero) is used, that means the `navigationPane` component will take the metadata from the first level under the `menu` element for rendering by the `commandNavigationItem` component.

4. In the Properties window, expand the **Data** section and set the following:
 - **Value**: Set to the menu model managed bean that is configured for the root `XMLMenuModel` class in the `faces-config.xml` file.

 **Note:**

The `value` attribute can reference root menu models and menu models referenced by shared nodes. If you reference a shared node in the `value` attribute, the `faces-config.xml` file needs to have a new managed bean entry with a different managed bean name than the one which is used in a root menu model definition in the menu model metadata file. This promotes the menu model of a shared node to a root menu model which can then be referred to in the `value` attribute.

- **Var:** Set to text that you will use in the `commandNavigationItem` components to get the needed data from the menu model.

As the hierarchy is created at runtime, and each node is stamped, the data for the current node is copied into the `var` attribute, which can then be addressed using an EL expression. You specify the name to use for this property in the EL expression using the `var` property.

 **Tip:**

You use the same value for the `var` attribute for every `navigationPane` component on the page or in the application.

5. In the Components window, from the Layout panel, in the Interactive Containers and Headers group, drag and drop a **Navigation Item** to the `nodeStamp` facet of the `navigationPane` component.
6. Set the values for the remaining attributes that have corresponding values in the metadata using EL expressions that refer to the menu model (whose metadata contains that information). You access these values using the value of the `var` attribute you set for the parent `navigationPane` component in Step 4 along with the name of the corresponding `itemNode` element that holds the value in the metadata. Table 20-6 shows the attributes on the navigation item that has corresponding values in the metadata.

Table 20-6 Navigation Item Attributes and the Associated Menu Model Attributes

Navigation Item Attribute	Associated Menu Model Element Attribute
text	label
action	doAction
icon	icon
destination	destination
visible	visible
rendered	rendered

For example, if you had set the `var` attribute on the parent `navigationPane` component to `menuNode`, you would use `#{menuNode.doAction}` as the EL expression for the value of the `action` attribute. This would resolve to the action

property set in the metadata for each node. [Example 20-3](#) shows the JSF code for binding to a menu model that has four levels of hierarchical nodes.

Example 20-3 Binding to the XMLMenuModel

```
<af:form>
  <af:navigationPane hint="buttons" level="0" value="#{root_menu}"
    var="menuNode">
    <f:facet name="nodeStamp">
      <af:commandNavigationItem text="#{menuNode.label}"
        action="#{menuNode.doAction}"
        icon="#{menuNode.icon}"
        destination="#{menuNode.destination}"/>
    </f:facet>
  </af:navigationPane>
  <af:navigationPane hint="tabs" level="1" value="#{root_menu}"
    var="menuNode">
    <f:facet name="nodeStamp">
      <af:commandNavigationItem text="#{menuNode.label}"
        action="#{menuNode.doAction}"
        icon="#{menuNode.icon}"
        destination="#{menuNode.destination}"/>
    </f:facet>
  </af:navigationPane>
  <af:navigationPane hint="bar" level="2" value="#{root_menu}"
    var="menuNode">
    <f:facet name="nodeStamp">
      <af:commandNavigationItem text="#{menuNode.label}"
        action="#{menuNode.doAction}"
        icon="#{menuNode.icon}"
        destination="#{menuNode.destination}"/>
    </f:facet>
  </af:navigationPane>
  <af:navigationPane hint="list" level="3" value="#{root_menu}"
    var="menuNode">
    <f:facet name="nodeStamp">
      <af:commandNavigationItem text="#{menuNode.label}"
        action="#{menuNode.doAction}"
        icon="#{menuNode.icon}"
        destination="#{menuNode.destination}"/>
    </f:facet>
  </af:navigationPane>
</af:form>
```

Note:

For information about how to let users close navigation tabs, see [What You May Need to Know About Removing Navigation Tabs](#).

How to Use the breadcrumbs Component with a Menu Model

Creating a breadcrumb using the menu model is similar to creating the page hierarchy; you use the `breadcrumbs` component with a `nodeStamp` facet that stamps a `commandNavigationItem` component with data from the model.

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Using a Menu Model to Create a Page Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create a breadcrumb using a menu model:

1. In the Components window, from the General Controls panel, in the Location group, drag and drop a **BreadCrumbs** onto the JSF page.
2. By default, breadcrumb links display in a horizontal line. To change the layout to be vertical, in the Properties window, expand the **Common** section and select vertical from the **Orientation** dropdown list.
3. In the Properties window, expand the **Data** section and the set the following:
 - **Value:** Set to the root `XMLMenuModel` managed bean as configured in the `faces-config.xml` file. This is the same bean to which the `navigationPane` component is bound.

 **Note:**

The `value` attribute should reference only a root menu model and not any menu models referenced through shared nodes. For example, if you use a shared node in your main `XMLMenuModel` element, JDeveloper would have created managed bean configurations for the shared node and the root `XMLMenuModel` bean that consumes the shared model. The shared model managed bean is automatically incorporated into the root menu model managed bean as the menu tree is parsed at startup.

- **Var:** Set to text that you will use in the `commandNavigationItem` components to get the needed data from the menu model.

As the hierarchy is created at runtime, and each node is stamped, the data for the current node is copied into the `var` attribute, which can then be addressed using an EL expression. You specify the name to use for this property in the EL expression using the `var` property.

 **Tip:**

You can use the same value for the `var` attribute for the `breadCrumbs` component as you did for the `navigationPane` components on the page or in the application.

4. In the Components window, from the Layout panel, in the Interactive Containers and Headers group, drag a **Navigation Item** and drop it inside the `nodeStamp` facet of the `breadCrumbs` component.

 **Note:**

The `nodeStamp` facet of the `breadCrumbs` component determines what links appear according to the menu model that you specify for the `value` attribute of the `breadCrumbs` component. If you do not specify the menu model you want to render for the `value` attribute of the `breadCrumbs` component, no links appear at runtime. Do not use a `nodeStamp` facet for the `breadCrumbs` component if you do not use a menu model because no stamps will be required.

5. Set the values for the remaining attributes that have corresponding values in the metadata using EL expressions that refer to the menu model (whose metadata contains that information). You access these values using the value of the `var` attribute you set for the parent `breadCrumbs` component in Step 3 along with the name of the corresponding `itemNode` element that holds the value in the metadata. [Table 20-6](#) shows the attributes on the navigation item that has corresponding values in the metadata.

For example, if you had set the `var` attribute on the `breadCrumbs` component to `menuNode`, you would use `#{menuNode.doAction}` as the EL expression for the value of the `action` attribute. This would resolve to the action property set in the metadata for each node.

```
<af:breadCrumbs var="menuNode" value="#{root_menu}">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
      action="#{menuNode.doAction}" />
  </f:facet>
</af:breadCrumbs>
```

How to Use the `menuBar` Component with a Menu Model

As described in [Using Menus, Toolbars, and Toolboxes](#), the `menuBar` and `menu` components are usually used to organize and create menus that users click to cause some change or action in the application. Where applicable, the `menuBar` component can be used with an `XMLMenuModel` implementation and managed beans to create a page hierarchy. Like the `breadCrumbs` or `navigationPane` component, when `menuBar` is bound to the root `XMLMenuModel` managed bean, you use one `commandNavigationItem` component in the `nodeStamp` facet to dynamically provide the menu items for navigating the page hierarchy.

When the page hierarchy of a website cannot be sufficiently represented by a tabbed navigation system (through a `navigationPane` or `panelTabbed` component), use the `menuBar` component to provide a navigation bar of menus and submenus. For example, a web store application with many shopping categories for users to browse might benefit from a horizontal arrangement of top-level menus in a bar instead of rendering all the categories and subcategories within tabs, subtabs or bars, and lists. With a `menuBar` bound to a menu model, submenus appear only when the user places the mouse cursor over a top-level menu or a submenu item. Not only does this arrangement reduce screen real estate, but the user can also quickly navigate from the top of the hierarchy to a page at the lowest level with just one click.

Unlike a `menuBar` component that is not bound to a menu model, a `menuBar` that is bound to a menu model is not detachable, and should not be used with a toolbar. Also,

do not use navigation tabs with a `menuBar` bound to a menu model on the same page. If you must use both, always place the `menuBar` component above the navigation tabs. You can, however, use a `menuBar` bound to a menu model with a `breadcrumbs` component bound to the same model on those pages where you want to show breadcrumb links.

If you want the menu bar to be styled, create a `decorativeBox` component by dragging and dropping a **Decorative Box** from the Layout panel of the Components window to the JSF page. Set the theme to determine how you want the tabs to appear. Valid values are:

- `default`
- `light`
- `medium`
- `dark`

Each value describes the look and feel applied to the application when you specify the theme value for the component. The application must use the Skyros skin or a skin that extends from the Skyros skin. The Alta skin does not use themes. You can change how the themes display. See [Customizing the Appearance Using Styles and Skins](#).

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Using a Menu Model to Create a Page Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create a horizontal menu bar using a menu model:

1. In the Components window, from the Menus and Toolbars panel, drag and drop a **Menu Bar** onto the JSF page.
2. In the Properties window, expand the **Menu Model** section and set the following values:
 - **Value:** Set to the root `XMLMenuModel` managed bean as configured in the `faces-config.xml` file. This is the same bean to which the `breadcrumbs` component is bound.

 **Note:**

The `value` attribute should reference only a root menu model and not any menu models referenced through shared nodes. For example, if you use a shared node in your main `XMLMenuModel` element, JDeveloper would have created managed bean configurations for the shared node and the root `XMLMenuModel` bean that consumes the shared model. The shared model managed bean is automatically incorporated into the root menu model managed bean as the menu tree is parsed at startup.

- **Var:** Set to the text that you will use in the `commandNavigationItem` components to get the needed data from the menu model.

As the hierarchy is created at runtime, and each node is stamped, the data for the current node is copied into the `var` attribute, which can then be addressed using an EL expression. You specify the name to use for this property in the EL expression using the `var` property.

 **Tip:**

You can use the same value for the `var` attribute for the `menuBar` component as you did for the `breadcrumbs` component on the page or in the application.

3. In the Components window, from the Layout panel, in the Interactive Containers and Headers group, drag and drop a **Navigation Item** to the `nodeStamp` facet of the `menuBar` component.

 **Note:**

The `nodeStamp` facet of the `menuBar` component determines what links appear according to the menu model that you specify for the `value` attribute of the `menuBar` component. If you do not specify the menu model you want to render for the `value` attribute of the `menuBar` component, no menu items will appear at runtime.

4. Set the values for the remaining attributes that have corresponding values in the metadata using EL expressions that refer to the menu model (whose metadata contains that information). You access these values using the value of the `var` attribute you set for the parent `menuBar` component in Step 2 along with the name of the corresponding `itemNode` element that holds the value in the metadata. [Table 20-6](#) shows the attributes on the navigation item that has corresponding values in the metadata.

For example, if you had set the `var` attribute on the `menuBar` component to `menuNode`, you would use `#{menuNode.doAction}` as the EL expression for the value of the `action` attribute. This would resolve to the action property set in the metadata for each node.

```
<af:menuBar var="menuNode" value="#{root_menu}">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
                             action="#{menuNode.doAction}"/>
  </f:facet>
</af:menuBar>
```

What Happens at Runtime: How the Menu Model Creates a Page Hierarchy

The `value` attribute of the menu model bound component (`navigationPane`, `breadcrumbs`, or `menuBar`) references the managed bean for the `XMLMenuModel` element. When that managed bean is requested, the following takes place:

- The `setSource()` method of the `XMLMenuModel` class is called with the location of the model's metadata, as specified in the `managed-property` element in the `faces-config.xml` file.
- An `InputStream` object to the metadata is made available to the parser (`SAXParser`); the metadata for the navigation items is parsed, and a call to `MenuContentHandler` method is made.
- The `MenuContentHandler` builds the navigation menu tree structure as a `List` object in the following manner:
 - The `startElement()` method is called at the start of processing a node in the metadata.
 - The `endElement()` method is called at the end of processing the node.
 - As each node is processed, a `List` of navigation menu nodes that make up the page hierarchy of the menu model is created.
- A `TreeModel` object is created from the list of navigation menu nodes.
- The `XMLMenuModel` object is created from the `TreeModel` object.

If a `groupNode` element has more than one child id in its `idref` attribute, the following occurs:

- The IDs are processed in the order they are listed. If no child node is found with the current ID, the next ID is used, and so on.
- Once a child node is found that matches the current ID in the `idref` list, then that node is checked to see if its `rendered` attribute is set to `true`, its `disabled` attribute is set to `false`, its `readOnly` attribute is set to `false`, and its `visible` attribute is set to `true`. If any of the criteria is not met, the next ID in the `idref` list is used, and so on.
- The first child node that matches the criteria is used to obtain the action outcome or destination URI. If no child nodes are found that match the criteria, an error is logged. However, no error will be shown in the UI.
- If the first child node that matches the criteria is another `groupNode` element, the processing continues into its children. The processing stops when an `itemNode` element that has either an `action` or `destination` attribute is encountered.
- When the `itemNode` element has an `action` attribute, the user selection initiates a `POST` action and the navigation is performed through the action outcome. When the `itemNode` element has a `destination` attribute, the user selection initiates a `GET` action and navigation is performed directly using the `destination` value.

The `XMLMenuModel` class provides the model that correctly highlights and enables the items on the navigation menus (such as tabs and bars) as you navigate through the navigation menu system. The model is also instantiated with values for `label`, `doAction`, and other properties that are used to dynamically generate the navigation items.

The `XMLMenuModel` class does no rendering; the model bound component uses the return value from the call to the `getFocusRowKey()` method to render the navigation menu items for a level on a page.

The `commandNavigationItem` component housed within the `nodeStamp` facet of the menu model bound component provides the label text and action outcome for each navigation item. Each time the `nodeStamp` facet is stamped, the data for the current navigation item is copied into an EL-reachable property, the name of which is defined

by the `var` attribute on the `navigationPane` component that houses the `nodeStamp` facet. The `nodeStamp` displays the data for each item by getting further properties from the EL-reachable property. Once the navigation menu has completed rendering, this property is removed (or reverted back to its previous value). When users select a navigation item, the default JSF `actionListener` mechanism uses the action outcome string or destination URI to handle the page navigation.

The `XMLMenuModel` class, in conjunction with `nodeStamp` facet also controls whether or not a navigation item is rendered as selected. As described earlier, the `XMLMenuModel` object is created from a tree model, which contains `viewId` attribute information for each node. The `XMLMenuModel` class has a method `getFocusRowKey()` that determines which page has focus, and automatically renders a node as selected if the node is on the focus path. The `getFocusRowKey()` method in its most simplistic fashion does the following:

- Gets the current `viewId` attribute.
- Compares the `viewId` attribute value with the IDs in internal maps used to resolve duplicate `viewId` values and in the `viewIdFocusPathMap` object that was built by traversing the tree when the menu model was created.
- Returns the focus path to the node with the current `viewId` attribute or returns `null` if the current `viewId` attribute value cannot be found.

The `viewId` attribute of a node is used to determine the focus `rowKey` object. Each item in the model is stamped based on the current `rowKey` object. As the user navigates and the current `viewId` attribute changes, the focus path of the model also changes and a new set of navigation items is accessed.

What You May Need to Know About Using Custom Attributes

Custom attributes that you have created can be displayed, but only for `itemNode` and `groupNode` elements.

Consider the `groupNode` code shown in the following example. Here, the `cardIcon` custom attribute is used in `groupNode` that can be bound to an alternate UI Widget. This model can be used by two UI Widget screens displaying the same model and using two different properties such as, `icon` and `cardIcon`.

[Example 20-4](#) shows an example of `cardIcon` custom attribute in `groupNode`.

Example 20-4 Custom Attribute for `groupNode` in `XMLMenuModel`

```
<groupNode
    id="groupNode_risk_management_tools"
    idref="_groupNode_risk_management_tools_"

label="#{menuBundle['oracle.apps.menu.ResourcesAttrBundle'].RISK_MANAGEMENT_TOOLS}">
    <itemNode
id="itemNode_risk_management_tools_setup_and_administration"

label="#{menuBundle['oracle.apps.menu.ResourcesAttrBundle'].SETUP_AND_ADMINISTRATION}"
"
        icon="/images/qual_gears_l6.png"
        cardIcon="gears.png"
        taskMenuSource="/WEB-INF/oracle/apps/grc/fuseplus/ui/menu/
SetupAndAdministration_taskmenu.xml"
        securedResourceName="/WEB-INF/oracle/apps/grc/ui/setup/view/setup/flow/
SetupMaintenancePGFlow.xml#SetupMaintenancePGFlow"
```

```

        focusViewId="/FuseOverview"
        webApp="GrcCore"
        applicationStripe="fscm"
        showOnHomeGrid="true"/>
        <itemNode id="itemNode_risk_management_tools_perspectives"

label="#{menuBundle['oracle.apps.menu.ResourcesAttrBundle'].PERSPECTIVES}"
        icon="/images/qual_folio_16.png"
        cardIcon="folio.png"
        taskMenuSource="/WEB-INF/oracle/apps/grc/fuseplus/ui/menu/
PerspectiveManagement_taskmenu.xml"
        securedResourceName="/WEB-INF/oracle/apps/grc/ui/perspective/view/flow/
ManagePerspectivesPGFlow.xml#ManagePerspectivesPGFlow"
        focusViewId="/FuseOverview"
        webApp="GrcCore"
        applicationStripe="fscm"
        showOnHomeGrid="true"/>
        <itemNode id="itemNode_risk_management_tools_surveys"
label="#{menuBundle['oracle.apps.menu.ResourcesAttrBundle'].SURVEYS}"
        icon="/images/qual_peopleconnect_16.png"
.....

```

Creating a Simple Navigational Hierarchy

ADF Faces allows you to create a simple hierarchical navigational model by using `commandNavigationItem` child components with `navigationPane` components. If you want to create a complex hierarchical model, then you must use a menu model.

Note:

If the application hierarchy is complex and consists of deeply nested pages, it is more efficient to use a menu model to create your navigation system. For details, see [Using a Menu Model to Create a Page Hierarchy](#).

[Using Navigation Items for a Page Hierarchy](#) describes a simple page hierarchy with three levels of links under a top-level root node, Home. [Figure 20-16](#) and [Figure 20-17](#) show an example of what the user interface could look like when the `navigationPane` component and individual `commandNavigationItem` components are used to create a view for the page hierarchy shown in [Figure 20-10](#).

Figure 20-16 Navigation Items Available from the View Employee Page



When you create the hierarchy manually, first determine the focus path of each page (that is, where exactly in the hierarchy the page resides) in order to determine the exact number of `navigationPane` and `commandNavigationItem` components needed for each page, as well as to determine the components that should be configured as selected when the user visits the page. For example, in [Figure 20-16](#), which shows the View Employee page, you would need three `navigationPane` components. In addition to the first-level tabs, only the second-level child bars of Employee Data are needed, and only the Employee Data tab and View Employee bar render as selected.

Similarly in [Figure 20-17](#), which shows the Health page, only the child bars of Benefits are needed, and the Benefits tab and Insurance bar must be configured as selected. Additionally for this page, you would create the child nodes under Insurance, which can be presented as a vertical list on the side of the page. The Health item in the vertical list is configured as selected, and the contents of the Health page are displayed in the middle, to the right of the vertical list.

Figure 20-17 Navigation Items Available from the Health Page



Regardless of the type of navigation items you use (such as tabs, bars or lists), you use a `navigationPane` component to represent one level of hierarchical links, and a series of `commandNavigationItem` child components within each `navigationPane` component to provide the actual navigation items. For example, in [Figure 20-17](#) the actual links for the first-level tabs (Benefits and Employee Data), the second-level bars (Insurance and Paid Time Off), and the Health and Dental links in the list are each provided by a `commandNavigationItem` component. Underneath the bars, to provide the breadcrumb links that show the focus path of the current page, you use a `breadcrumbs` component with the required number of child `commandNavigationItem` components.

How to Create a Simple Page Hierarchy

When your navigational hierarchy contains only a few pages and is not very deep, you can choose to manually create the hierarchy. Doing so involves creating the navigation rule and navigation cases, using the `navigationPane` component to create the hierarchy, and using the `commandNavigationItem` component to create the links.

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Creating a Simple Navigational Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To manually create a navigation hierarchy:

1. In the Applications window, expand the **WEB-INF** node and double-click **faces-config.xml**.
2. In the source editor, create one global JSF navigation rule that has the navigation cases for all the nodes (that is, pages) in the page hierarchy.

For example, the page hierarchy shown in [Figure 20-10](#) has 10 nodes, including the global Help node. Thus, you would create 10 navigation cases within one global navigation rule in the `faces-config.xml` file, as shown in the following example.

For each navigation case, specify a unique outcome string, and the path to the JSF page that should be displayed when the navigation system returns an outcome value that matches the specified string.

```
<navigation-rule>
  <navigation-case>
    <from-outcome>goHome</from-outcome>
    <to-view-id>/home.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHelp</from-outcome>
    <to-view-id>/globalhelp.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goEmp</from-outcome>
    <to-view-id>/empdata.jsf</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goBene</from-outcome>
    <to-view-id>/benefits.jsf</to-view-id>
  </navigation-case>
```

```
<navigation-case>
  <from-outcome>goIns</from-outcome>
  <to-view-id>/insurance.jsf</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>goPto</from-outcome>
  <to-view-id>/pto.jsf</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>goView</from-outcome>
  <to-view-id>/viewdata.jsf</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>goCreate</from-outcome>
  <to-view-id>/createemp.jsf</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>goHealth</from-outcome>
  <to-view-id>/health.jsf</to-view-id>
</navigation-case>
<navigation-case>
  <from-outcome>goDental</from-outcome>
  <to-view-id>/dental.jsf</to-view-id>
</navigation-case>
</navigation-rule>
```

For information about creating navigation cases in JDeveloper, see [Defining Page Flows](#).

3. Create the JSF pages for all the hierarchical nodes. If you want the navigation tabs to be styled, create a `decorativeBox` component by dragging and dropping a **Decorative Box** from the Layout panel of the Components window to each page. Set the theme to determine how you want the tabs to appear. Valid values are:
 - default
 - light
 - medium
 - dark

Each value describes the look and feel applied to the application when you specify the theme value for the component. The application must use the Skyros skin or a skin that extends from the Skyros skin. The Alta skin does not use themes. You can change how the themes display. See [Customizing the Appearance Using Styles and Skins](#). To consider using a page template to achieve the positioning and visual styling of your JSF pages, see [Using Page Templates](#).

4. In the Components window, from the Layout panel, in the Interactive Containers and Headers group, drag and drop a **Navigation Pane** to each page. Drag and drop a `navigationPane` component for each level of the hierarchy on the page.

For example, to create the Health page, as shown in [Figure 20-17](#), drag and drop four `navigationPane` components. In the Health page, the components are dropped into specific areas of a template that already contains layout components to create the look and feel of the page.

5. For each `navigationPane` component, in the Properties window, expand the **Common** section and select one of the following types of navigation items from the **Hint** dropdown list to determine how the `navigationPane` component displays:

- **bar**: Displays the navigation items separated by a bar, for example the **Insurance** and **Paid Time Off** links in [Figure 20-17](#).
 - **buttons**: Displays the navigation items separated by a bar in a global area, for example the **Home** and **Help** links in [Figure 20-17](#).
 - **choice**: Displays the navigation items in a popup list when the associated dropdown icon is clicked. You must include a value for the `navigationPane` component's `icon` attribute and you can associate a label to the dropdown list using `title` attribute.
 - **list**: Displays the navigation items in a bulleted list, for example the **Health** and **Dental** links in [Figure 20-17](#).
 - **tabs**: Displays the navigation items as tabs, for example the **Benefits** and **Employee Data** tabs in [Figure 20-17](#).
6. For each `navigationPane` component, add the needed `commandNavigationItem` components to represent the different links by dragging and dropping a **Navigation Item** from the Interactive Containers and Headers group in the Layout panel of the Components window. Drop a **Navigation Item** as a child to the `navigationPane` component for each link needed.

For example, to create the Health page as shown in [Figure 20-17](#), you would use a total of eight `commandNavigationItem` components, two for each `navigationPane` component.

Performance Tip:

At runtime, when available browser space is less than the space needed to display the contents in a tab or bar of a navigation pane, or the contents of the breadcrumb, ADF Faces automatically displays overflow icons that enable users to select and navigate to those items that are out of view. The number of child components within a `navigationPane` or `breadcrumbs` component, and the complexity of the children, will affect the performance of the items within the overflow. You should set the size of the `navigationPane` or `breadcrumbs` component to avoid overflow when possible.

7. For each `commandNavigationItem` component, set the navigation to the desired page. In the Properties window, expand the **Common** section and provide a static string outcome of an action or use an EL expression to reference an action method in the **Action** field. If you use a string, it must match the navigation metadata set up in the navigation rules for the page created in [Step 2](#). If referencing a method, that method must return the required string.
8. In the Properties window, expand the **Behavior** section and select `true` from the **Selected** dropdown list if the `commandNavigationItem` component should be displayed as selected when the page is first rendered, and `false` if it should not.

At runtime, when a navigation item is selected by the user, that component's `selected` attribute changes to `selected="true"` and the appearance changes to indicate to the user that the item has been selected. For example, in [Figure 20-17](#) the Benefits tab, Insurance bar, and Health list item are shown as selected by a change in either background color or font style. You do not have to write any code to show the selected status; the `selected` attribute on the `commandNavigationItem`

component for that item takes care of turning on the selected status when the attribute value is `true`.

Example 20-5 Using `actionListener` to Change Selected State

```
<!-- JSF Page Code ----->
<af:navigationPane hint="tabs">
  <af:commandNavigationItem text="Benefits"
    actionListener="#{myBean.navigationItemAction}"
    partialSubmit="true"../>
  .
</af:navigationPane>

<!-- Managed Bean Code ----->
public void navigationItemAction(ActionEvent event)
{
  UIComponent actionItem = event.getComponent();
  UIComponent parent = actionItem.getParent();
  while (! (parent instanceof UIXNavigationHierarchy) )
  {
    parent = parent.getParent();
    if (parent == null)
    {
      System.err.println(
        "Unexpected component hierarchy, no UIXNavigationHierarchy found.");
      return;
    }
  }

  List<UIComponent> children = parent.getChildren();
  for (UIComponent child : children)
  {
    FacesBean childFacesBean = ((UIXComponent) child).getFacesBean();
    FacesBean.Type type = childFacesBean.getType();
    PropertyKey selectedKey = type.findKey("selected");
    if (selectedKey != null)
    {
      childFacesBean.setProperty(selectedKey, child==actionItem);
    }
  }
}

RequestContext adfContext = RequestContext.getCurrentInstance();
adfContext.addPartialTarget(parent);
}
```

The following example shows code used to generate the navigation items that are available when the current page is Health. Because the Health page is accessed from the Insurance page from the Benefits page, the `commandNavigationItem` components for those three links have `selected="true"`.

```
<af:navigationPane hint="buttons">
  <af:commandNavigationItem text="Home" action="goHome"/>
  <af:commandNavigationItem text="Help" action="goHelp"/>
</af:navigationPane>
...
<af:navigationPane hint="tabs">
  <af:commandNavigationItem text="Benefits" action="goBene"
    selected="true"/>
  <af:commandNavigationItem text="Employee Data" action="goEmp"/>
</af:navigationPane>
...
<af:navigationPane hint="bar">
```

```

    <af:commandNavigationItem text="Insurance" action="goIns"
        selected="true" />
    <af:commandNavigationItem text="Paid Time Off" action="goPto" />
</af:navigationPane>
...
<af:navigationPane hint="list">
    <af:commandNavigationItem text="Health" action="goHealth"
        selected="true" />
    <af:commandNavigationItem text="Dental" action="goDental" />
</af:navigationPane>

```

To change the selected state programmatically, you have to write a backing bean method to handle an action event. Then reference the method on the `actionListener` attribute of the `commandNavigationItem` components, as shown in [Example 20-5](#).

How to Use the `breadcrumbs` Component

In both [Figure 20-16](#) and [Figure 20-17](#), the user's current position in the page hierarchy is indicated by a path of links from the current page back to the root page. The path of links, also known as breadcrumbs, is displayed beneath the secondary bars, above the vertical lists (if any). To manually create such a path of links, you use the `breadcrumbs` component with a series of `commandNavigationItem` components as children.

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. See [Creating a Simple Navigational Hierarchy](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To manually create a breadcrumb:

1. In the Components window, from the General Controls panel, in the Location group, drag and drop a **BreadCrumbs** onto the JSF page.
2. By default, breadcrumb links are displayed in a horizontal line. To change the layout to be vertical, in the Properties window, expand the **Common** section and select `vertical` from the **Orientation** dropdown list.
3. For each link in the breadcrumb, create a `commandNavigationItem` component by dragging and dropping a **Navigation Item** from the Interactive Containers and Headers group in the Layout panel of the Components window as a child to the `breadcrumbs` component. The last item should represent the current page.

Tip:

Depending on the renderer or client device type, the last link in the breadcrumb may not be displayed, but you still must add the `commandNavigationItem` component for it. On clients that do display the last breadcrumb link, the link is always disabled automatically because it corresponds to the current page.

4. For each `commandNavigationItem` component (except the last), set the navigation to the desired page. In the Properties window, expand the **Common** section and provide a static string outcome of an action or use an EL expression to reference

an action method in the **Action** field. If you use a string, it must match the navigation metadata set up in the navigation rule for the page created in Step 2 as described in [How to Create a Simple Page Hierarchy](#). If referencing a method, that method must return the required string.

Example 20-6 BreadCrumbs Component With Individual CommandNavigationItem Children

```
<af:breadcrumbs>
  <af:commandNavigationItem text="Home" action="goHome" />
  <af:commandNavigationItem text="Benefits" action="goBene" />
  <af:commandNavigationItem text="Insurance" action="goIns" />
  <af:commandNavigationItem text="Health" />
</af:breadcrumbs>
```

For example, to create the breadcrumb as shown on the Health page in [Figure 20-17](#), drag and drop four `commandNavigationItem` components, as shown in [Example 20-6](#).

What You May Need to Know About Removing Navigation Tabs

You can configure a `navigationPane` component whose `hint` attribute value is `tabs` so that the individual tabs can be closed. You can set it such that all tabs can be closed, all but the last tab can be closed, or no tabs can be closed. When navigation tabs are configured to be removed, a close icon (for example, an X) displays at the end of each tab as the mouse cursor hovers over the tab.

To enable tabs removal in a `navigationPane` component when `hint="tabs"`, you need to do the following:

- Set the `itemRemoval` attribute on `navigationPane hint="tabs"` to `all` or `allExceptLast`. When set to `allExceptLast`, all but one tab can be closed. This means as a user closes tabs, when there is only one tab left, that single last tab cannot be closed.
- Implement a handler to do the tab removal. When a user closes a tab, an `ItemEvent` of type `remove` is launched. Your code must handle this event and the actual removal of the tab, and any other desired functionality (for example, show a warning dialog on how to handle child components). See [Handling Events](#). For information about using popup dialogs and windows, see [Using Popup Dialogs, Menus, and Windows](#).
- Set the `itemListener` attribute on the `commandNavigationItem` component to an EL expression that resolves to the handler method that handles the actual tab removal, as shown in [Example 20-7](#).

Example 20-7 Using itemListener to Remove a Tab Item

```
<!-- JSF Page Code -->
<af:navigationPane hint="tabs" itemRemoval="all">
  <af:commandNavigationItem text="Benefits" partialSubmit="true"
    itemListener="#{closebean.handleCloseTabItem}" />
  ...
</af:navigationPane>

// Managed Bean Code
import oracle.adf.view.rich.event.ItemEvent;
...
public void handleCloseTabItem(ItemEvent itemEvent)
{
  if (itemEvent.getType().equals(ItemEvent.Type.remove))
```

```

{
  Object item = itemEvent.getSource();
  if (item instanceof RichCommandNavigationItem)
  {
    RichCommandNavigationItem tabItem = (RichCommandNavigationItem) item;
    tabItem.setVisible(false);
    // do other desired functionality here ...
  }
}
}

```

What You May Need to Know About the Size of Navigation Tabs

By default, the size of the tabs rendered by a `navigationPane` component that has its `hint` attribute value set to `tabs` is determined by the length of the text used as the label. You can configure the size of these tabs by configuring the following attributes for the `navigationPane` component:

- `maxTabSize`: Set to a size in pixels. The tabs will never be larger than this size.
- `minTabSize`: Set to a size in pixels. The tabs will never be smaller than this size.
- `truncationStyle`: Set to `ellipsis` if you want an ellipsis to display after truncated text that cannot fit, based on the `maxTabSize`. If set to `none`, then if the text does not fit on the tab, it will simply be truncated.

What You May Need to Know About Skinning and Navigation Tabs

You can use the `-tr-layout-type` skinning key to configure the type of indicator that the `navigationPane` component renders in an application window that is in a compressed layout. That is, the application window is not wide enough to display all the navigation tabs.

Figure 20-18 shows an overflow indicator that renders a dropdown list where the user can choose the navigation tab to navigate to.

Figure 20-18 Overflow Indicator for a `navigationPane` Component in Compressed Layout



Example 20-8 shows how you configure the `-tr-layout-type` skinning key so that the `navigationPane` component displays an overflow indicator.

Rather than display overflow indicators, as shown in Figure 20-18, you can configure the `-tr-layout-type` skinning key for the `navigationPane` component so that the component renders a conveyor belt where users can scroll left or right to tabs that are not currently visible. Configuring the `-tr-layout-type` skinning key also renders all navigation tabs in one dropdown list, as shown in Figure 20-19. This configuration only takes effect if the `navigationPane` component's `hint` attribute is set to `tabs`. If the

navigationPane component's hint attribute is set to another value, set the `-tr-layout-type` skinning key to `overflow`.

[Example 20-9](#) shows how you configure the `-tr-layout-type` skinning key so that the navigationPane component renders a conveyor belt.

[Figure 20-19](#) shows the navigationPane component rendering a conveyor belt in a compressed layout.

Figure 20-19 Conveyor Belt for a navigationPane Component in Compressed Layout



See [Customizing the Appearance Using Styles and Skins](#).

Example 20-8 Using a Skinning Key to Set the Compressed Layout to Overflow

```
af|navigationPane {
  -tr-layout-type: overflow;
}
```

Example 20-9 Using a Skinning Key to Set the Compressed Layout to Conveyor Belt

```
af|navigationPane {
  -tr-layout-type: conveyor;
}
```

Using Train Components to Create Navigation Items for a Multistep Process

ADF Faces provides a train model which allows you to display a series of pages in a particular order and indicate the location of the current step in the navigation. The train model allows users to navigate only in the order that you have set for display.

Note:

If your application uses the Fusion technology stack or ADF Controller, then you should use ADF task flows to create the navigation system for your application page hierarchy. See [Using Train Components in Bounded Task Flows](#) section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

If you have a set of pages that users should visit in a particular order, consider using the `train` component on each page to display a series of navigation items that guide users through the multistep process. Figure 20-20 shows an example of what a rendered `train` component looks like on a page. Not only does a `train` component display the number of steps in a multistep process, it also indicates the location of the current step in relation to the entire process.

Figure 20-20 Navigation Items Rendered by a `train` Component



The `train` component renders each configured step represented as a train stop, and with all the stops connected by lines. Each train stop has an image (for example, a square block) with a label underneath the image.

Each train stop corresponds to one step or one page in your multistep process. Users navigate the train stops by clicking an image or label, which causes a new page to display. Typically, train stops must be visited in sequence, that is, a user must start at Step 1, move to Step 2, then Step 3, and so on; a user cannot jump to Step 3 if the user has not visited Step 2. Train stops can also be configured so that end users do not have to visit the stops in sequence. When you configure train stops in this way, all train stops that can be directly visited are enabled.

As shown in Figure 20-20, the `train` component provides at least four styles for train stops. The current stop where the user is visiting is indicated by a bold font style in the train stop's label, and a different image for the stop; visited stops before the current stop are indicated by a different label font color and image color; the next stop immediately after the current stop appears enabled; any other stops that have not been visited are grayed-out.

A train stop can include a subtrain, that is, you configure an action component (for example, a `button` component) to start a child multistep process from a parent stop, and then return to the correct parent stop after completing the subprocess. Suppose stop number 3 has a subprocess train containing two stops, when the user navigates into the first stop in the subprocess train, ADF Faces displays an icon representation of the parent train before and after the subprocess train, as shown in Figure 20-21.

Figure 20-21 Parent Train Icons At Start and End of a Subtrain



You can use the `trainButtonBar` component in conjunction with the `train` component to provide additional navigation items for the train, in the form of forward and backward arrows, as shown in Figure 20-22. These arrows allow users to navigate only to the next or previous train stop from the current stop. You can also use the `trainButtonBar` component without a `train` component. For example, you may want

to display just the forward and backward arrows without displaying the stops when not all of the stops will be visited based on some conditional logic.

 **Note:**

You need to add the following code to your `.css` file to view the forward and backward icons:

```
af|trainButtonBar  
{  
  
-tr-display-mode: icon;  
}
```

Figure 20-22 Navigation Buttons Rendered by a `trainButtonBar` Component



The `NavigationLayout` attribute is used to provide order from forward and backward arrows and `rangeText` (page information). The default value for `NavigationLayout`, is `auto` that only displays the forward and backward arrows on the page. It is possible to set other values such as `prevButton` `rangeText` `nextButton`, as shown in [Figure 20-23](#). If the `NavigationLayout` attribute is not set in the `jspx`, the default value will be assigned.

Figure 20-23 Navigation Buttons Rendered by a `trainButtonBar` Component with Page Information



Both train components work by having the `value` attribute bound to a train model of type `org.apache.myfaces.trinidad.model.MenuModel`. The train menu model contains the information needed to:

- Control a specific train behavior (that is, how the train advances users through the train stops to complete the multistep process).
- Dynamically generate the train stops, including the train stop labels, and the status of each stop (that is, whether a stop is currently selected, visited, unvisited, or disabled).

 **Note:**

In an application that uses the ADF Model layer and ADF Controller, this navigation and display is set up and handled in a different manner. See the "Using Train Components in Bounded Task Flows" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Briefly, a menu model for the train is implemented by extending the `MenuModel` abstract class, which in turn extends the `TreeModel` class (See [Using Tables, Trees, and Other Collection-Based Components](#)). A `MenuModel` object represents the menu structure of a page or application or could represent the hierarchy of pages and stops involved in a flow.

Because an instance of a `MenuModel` class is a special kind of a `TreeModel` object, the nodes in the `TreeModel` object can represent the stops of a train. The node instance that represents a train stop within the train component can be of type `TrainStopModel`, or it can be any object as long as it provides the same EL structure as a `TrainStopModel` object. However, the `TrainStopModel` class is a convenient interface that exposes all the relevant methods to retrieve the outcome, as well as the label of a stop and its `immediate`, `disabled`, and `visited` attribute states.

The `MenuModel` class can also indicate where in the tree the current train stop (page) is focused. The `getFocusRowKey()` method in the `MenuModel` class returns the `rowKey` object of the focus page for the current `viewId`. The menu model implementation for the train must also have a specific train behavior. You can implement this behavior by extending the `MenuModel` abstract class or the `ProcessMenuModel` convenience class. Both these classes come from the following package:

```
org.apache.myfaces.trinidad.model
```

The train behavior controls what other stops along the train users can visit while visiting the current train stop.

To create a train stop model, you can either extend the `TrainStopModel` abstract class and implement the abstract methods, or you can create your own class with the same method signatures. Your class must return a `rowData` object. An instance of this class represents a `rowData` object in the underlying collection (for the `MenuModel` implementation).

Binding a train component to a menu model is similar to binding a `navigationPane` component to an `XMLMenuModel` class using the `value` attribute (described in [How to Bind the navigationPane Component to the Menu Model](#)). However, as long as your `TrainStopModel` implementation represents a `rowData` object, you do not need to use the `nodeStamp` facet and its `commandNavigationItem` component to provide the train stops. At runtime ADF Faces dynamically creates the `nodeStamp` facet and `commandNavigationItem` component, and automatically binds the methods in the train stop model to the appropriate properties on the `commandNavigationItem` component. The following example shows the simplified binding for a train.

```
<af:train value="{simpleTrainModel}"/>
```

 **Tip:**

If you need to collate information for the train stops from various places, then you will need to manually create the `nodeStamp` facet and the individual `commandNavigationItem` components that represent the train stops. See [How to Bind to the Train Model in JSF Pages](#).

The `MenuModel` implementation of your train model must provide specific train behavior. Train behavior defines how you want to control the pages users can access based on the page they are currently visiting. ADF Faces supports two train behaviors: Plus One and Max Visited.

Suppose there are 5 pages or stops in a train, and the user has navigated from page 1 to page 4 sequentially. Currently the user is at page 4. Where the user can go next depends on which train behavior the train model implements:

- Plus One behavior: the user can go to page 3 or page 5
- Max Visited behavior: the user can visit pages 1 to 3 (previously visited) and page 5 because it is the next page in the sequence. If the user goes to page 2, the next page that the user can visit is page 1, 3 or 4. The user cannot visit page 5 because page 4 was the maximum visited train stop in the sequence.

To define and use a train for all pages in a multistep process:

- Create a JSF navigation rule and the navigation cases for the train. Creating a navigation rule and its navigation cases for a train is similar to [How to Create a Simple Page Hierarchy](#), where you create one global navigation rule that has the navigation cases for all the train stops in the train.

 **Note:**

You may want to set the value of the `redirect` element to `true` for each navigation case that you define within the JSF navigation rule if each train stop is an individual page and you want the client browser's URL to reference each new page. If you enable partial page rendering, the displayed URL may be different. For information about the `redirect` element, see the JavaServer Faces specification. For information about partial page rendering, see [Rerendering Partial Page Content](#).

- Create a train model that implements a specific train behavior and provides the train stop items for stamping. This includes creating a train stop model class and a menu model class. See [How to Create the Train Model](#).
- Configure managed beans for the train model. See [How to Configure Managed Beans for the Train Model](#).
- Create a JSF page for each train stop.
- On each page, bind the `train` component to the train model. See [How to Bind to the Train Model in JSF Pages](#). Optionally, bind the `trainButtonBar` component to the same train model if you want to provide additional navigation buttons for the train.

How to Create the Train Model

To define a train menu model, you create:

- A train stop model that provides data for rendering a train stop.
- A `MenuModel` implementation with a specific train behavior (like `Max Visited` or `Plus One`) that controls what stops along the train users can visit while visiting at a current train stop, which stops should be disabled or whether the train needs to be navigated sequentially or not, among other things.

ADF Faces makes it easier for you to define a train menu model by providing additional public classes, such as:

- The abstract class `TrainStopModel` for implementing a train stop model.
- The classes `ProcessMenuModel` and `ProcessUtils` that implement the `Max Visited` and `Plus One` behaviors.

Users can either implement their own custom train behavior by overriding `MenuModel` or extend the existing `ProcessMenuModel` to provide specialized behavior.

For examples of train model classes, see the `oracle.adfdemo.view.nav.rich` package of the ADF Faces Components Demonstration application.

Before you begin:

It may help to understand how a train component's attributes affect functionality. See [Using Train Components to Create Navigation Items for a Multistep Process](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To create the train model:

1. Create a train stop model class. A train stop model object holds the row data for stamping each train stop. The train stop model implementation you create should set and get the properties for each stop in the train, and define the methods required to render a train stop. The properties of a train stop correspond to the properties of the `commandNavigationItem` component. This will allow you to use the simplified binding (`<af:train value="#{simpleTrainModel}"/>`).

Alternatively, you can extend the abstract class `TrainStopModel`, and implement the abstract methods in the subclass.

The properties on the `commandNavigationItem` component that will be automatically EL bound are:

- `action`: A static action outcome or a reference to an action method that returns an action outcome. The outcome is used for page navigation through the default `ActionListener` mechanism in JSF.
- `disabled`: A boolean value that indicates whether or not the train stop should be noninteractive. Note that the train behavior you elect to use affects the value of this property. See [Step 2](#).
- `immediate`: A boolean value that determines whether or not data validations should be performed. Note that the train behavior you elect to use affects the value of this property. See [Step 2](#).

- `messageType`: A value that specifies a message alert icon over the train stop image. Possible values are `none`, `error`, `warning`, and `info`, and `complete`. For information about messages, see [Displaying Tips, Messages, and Help](#).
 - `shortDesc`: A value that is commonly used by client user agents to display as tooltip help text for the train stop.
 - `showRequired`: A boolean value that determines whether or not to display an asterisk next to the train stop to indicate that required values are contained in that train stop page.
 - `textAndAccessKey`: A single value that sets both the label text to display for the train stop, as well as the access key to use.
 - `visited`: A boolean value that indicates whether or not the train stop has already been visited. Note that the train behavior you elect to use affects the value of this property. See [Step 2](#).
2. Create a class based on the `MenuItem` class to facilitate the construction of a train model.

The `MenuItem` implementation of your train model must have a specific train behavior. The `ProcessMenuItem` class in the `org.apache.myfaces.trinidad.model` package is a reference implementation of the `MenuItem` class that supports the two train behaviors: `Plus One` and `Max Visited`. To implement a train behavior for a train model, you can either extend the `ProcessMenuItem` class, or create your own.

In your train model class, you override the `getFocusRowKey()` method (see the `MenuItem` class) and implement a train behavior (see the `ProcessMenuItem` and `ProcessUtils` classes).

The train behaviors provided in the `ProcessMenuItem` class have an effect on the `visited`, `immediate`, and `disabled` properties of the `commandNavigationItem` component.

The `visited` attribute is set to `true` only if that page in the train has been visited. The `ProcessMenuItem` class uses the following logic to determine the value of the `visited` attribute:

- **Max Visited**: A max visited stop is the farthest stop the user has visited in the current session. The `visited` attribute is set to `true` for any stop if it is before a max visited stop, or if it is the max visited stop itself.
- **Plus One**: A plus one stop does not keep track of the farthest stop that was visited. The `visited` attribute is set to `true` for the current stop, or a stop that is before the current stop.

When the data on the current page does not have to be validated, the `immediate` attribute should be set to `true`. Suppose page 4 in the `Plus One` behavior described earlier has data that must be validated. If the user has advanced to page 4 and then goes back to page 2, the user has to come back to page 4 again later to proceed on to page 5. This means the data on page 4 does not have to be validated when going back to page 1, 2, or 3 from page 4, but the data should be validated when going ahead to page 5. For information about how the `immediate` attribute works, see [Using the Immediate Attribute](#).

The `ProcessMenuItem` class uses the following logic to determine the value of the `immediate` attribute:

- **Plus One:** The `immediate` attribute is set to `true` for any previous step, and `false` otherwise.
- **Max Visited:** When the current page and the maximum page visited are the same, the behavior is the same as the Plus One scenario. If the current page is before the maximum page visited, then the `immediate` attribute is set to `false`.

 **Note:**

In an application that uses the ADF Model layer, the `pageDefinition` element in a page definition file supports an attribute (`SkipValidation`) that, when set to `true`, skips data validation for the page. Set `SkipValidation` to `true` if you want users to navigate from the page without invoking data validation. See `pageNamePageDef.xml` in *Developing Fusion Web Applications with Oracle Application Development Framework*.

The `disabled` attribute is set to `true` only if that page in the train cannot be reached from the current page. The `ProcessMenuModel` class uses the following logic to determine the value of the `disabled` attribute:

- **Plus One:** The `disabled` attribute will be `true` for any page beyond the next available page.
- **Max Visited:** When the current stop and the maximum page visited are the same, the behavior is the same as the Plus One behavior. If the current page is before the maximum page visited, then `disabled` is set to `true` for any page beyond the maximum page visited.

By default, ADF Faces uses the Max Visited behavior when a non-null `maxPathKey` value is passed into the train model, as determined by the managed bean you will create to support the behavior (see [How to Configure Managed Beans for the Train Model](#)). If the `maxPathKey` value is `null`, then ADF Faces uses the Plus One behavior.

How to Configure Managed Beans for the Train Model

You use managed beans in a train model to gather the individual train stops into an `ArrayList` object, which is turned into the tree model that is then injected into a menu model to bind to the `value` attribute of the train component. You must instantiate the beans with the proper values for injection into the models, and you also have to configure a managed bean for each train stop or page in the train.

Before you begin:

It may help to understand how a train component's attributes affect functionality. See [Using Train Components to Create Navigation Items for a Multistep Process](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To configure managed beans for the train model:

1. Configure a managed bean for each stop in the train, with values for the properties that require setting at instantiation, to create the train stops to pass into an `ArrayList`.

If a train stop has subprocess train children, there should be a managed bean for each subprocess train stop as well.

Each bean should be an instance of the train stop model class created in [How to Create the Train Model](#). [Example 20-10](#) shows sample managed bean code for train stops in the `faces-config.xml` file.

The managed properties set the values to the train stop model object (the class created in Step 1 in [How to Create the Train Model](#)).

The `viewId` value is the path and file name to the page that is navigated to when the user clicks a train stop.

The `outcome` property value is the action outcome string that matches a JSF navigation case. The default JSF `ActionListener` mechanism is used to choose the page associated with the train stop as the view to navigate to when the train stop is selected.

The `label` property value is the train stop label text that displays beneath the train stop image. The value can be static or an EL expression that evaluates to a string in a resource bundle.

The `model` property value is the managed bean name of the train model (see [Example 20-14](#)).

If a train stop has subprocess train children, the managed bean configuration should also include the property (for example, `children`) that lists the managed bean names of the subprocess train stops in value expressions (for example, `#{train4a}`), as shown in [Example 20-11](#).

2. Configure a managed bean that is an instance of an `ArrayList` object to create the list of train stops to pass into the train tree model.

[Example 20-12](#) shows sample managed bean code for creating the train stop list.

The `list-entries` element contains the managed bean names for the train stops (excluding subprocess train stops) in value expressions (for example, `#{train1}`), listed in the order that the stops should appear on the train.

3. Configure a managed bean to create the train tree model from the train list.

The train tree model wraps the entire train list, including any subprocess train lists. The train model managed bean should be instantiated with a `childProperty` value that is the same as the property name that represents the list of subprocess train children (see [Example 20-11](#)).

The `childProperty` property defines the property name to use to get the child list entries of each train stop that has a subprocess train.

The `wrappedData` property value is the train list instance to wrap, created by the managed bean in Step 2.

4. Configure a managed bean to create the train model from the train tree model.

This is the bean to which the `train` component on each page is bound. The train model wraps the train tree model. The train model managed bean should be instantiated with a `viewIdProperty` value that is the same as the property name that represents the pages associated with the train stops.

[Example 20-14](#) shows sample managed bean code for a train model.

The `viewIdProperty` property value is set to the property that is used to specify the page to navigate to when the user clicks the train stop.

The `wrappedData` property value is the train tree instance to wrap, created by the managed bean in Step 3.

The `maxPathKey` property value is the value to pass into the train model for using the Max Visited train behavior. ADF Faces uses the Max Visited behavior when a non-null `maxPathKey` value is passed into the train model. If the `maxPathKey` value is null, then ADF Faces uses the Plus One behavior.

Example 20-10 Managed Beans for All Train Stops

```
<!-- First train stop -->
<managed-bean>
  <managed-bean-name>train1</managed-bean-name>
  <managed-bean-class>project1.DemoTrainStopModel</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/train.jsf</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>guide.train</value>
  </managed-property>
  <managed-property>
    <property-name>label</property-name>
    <value>First Step</value>
  </managed-property>
  <managed-property>
    <property-name>model</property-name>
    <value>trainMenuModel</value>
  </managed-property>
</managed-bean>
...
<!-- Second train stop -->
<managed-bean>
  <managed-bean-name>train2</managed-bean-name>
  <managed-bean-class>project1.DemoTrainStopModel</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/train2.jsf</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>guide.train2</value>
  </managed-property>
  <managed-property>
    <property-name>label</property-name>
    <value>Second Step</value>
  </managed-property>
  <managed-property>
    <property-name>model</property-name>
    <value>trainMenuModel</value>
  </managed-property>
</managed-bean>
...
<!-- And so on -->
...
```

Example 20-11 Managed Bean for a Train Stop with Subprocess train Children

```

<managed-bean>
  <managed-bean-name>train4</managed-bean-name>
  <managed-bean-class>project1.DemoTrainStopModel</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>viewId</property-name>
    <value>/train4.jsf</value>
  </managed-property>
  <managed-property>
    <property-name>outcome</property-name>
    <value>guide.train4</value>
  </managed-property>
  <managed-property>
    <property-name>label</property-name>
    <value>Fourth Step</value>
  </managed-property>
  <managed-property>
    <property-name>children</property-name>
    <list-entries>
      <value-class>project1.DemoTrainStopModel</value-class>
      <value>#{train4a}</value>
      <value>#{train4b}</value>
      <value>#{train4c}</value>
    </list-entries>
  </managed-property>
  <managed-property>
    <property-name>model</property-name>
    <value>trainMenuModel</value>
  </managed-property>
</managed-bean>

```

Example 20-12 Managed Bean for Train List

```

<managed-bean>
  <managed-bean-name>trainList</managed-bean-name>
  <managed-bean-class>
    java.util.ArrayList
  </managed-bean-class>
  <managed-bean-scope>
    none
  </managed-bean-scope>
  <list-entries>
    <value-class>project1.DemoTrainStopModel</value-class>
    <value>#{train1}</value>
    <value>#{train2}</value>
    <value>#{train3}</value>
    <value>#{train4}</value>
    <value>#{train5}</value>
  </list-entries>
</managed-bean>

```

Example 20-13 Managed Bean for Train Tree Model

```

<managed-bean>
  <managed-bean-name>trainTree</managed-bean-name>
  <managed-bean-class>
    org.apache.myfaces.trinidad.model.ChildPropertyTreeModel
  </managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>

```

```

    <property-name>childProperty</property-name>
    <value>children</value>
  </managed-property>
</managed-property>
  <property-name>wrappedData</property-name>
  <value>#{trainList}</value>
</managed-property>
</managed-bean>

```

Example 20-14 Managed Bean for Train Model

```

<managed-bean>
  <managed-bean-name>trainMenuModel</managed-bean-name>
  <managed-bean-class>
    org.apache.myfaces.trinidad.model.ProcessMenuModel
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>viewIdProperty</property-name>
    <value>viewId</value>
  </managed-property>
  <managed-property>
    <property-name>wrappedData</property-name>
    <value>#{trainTree}</value>
  </managed-property>
  <!-- to enable plusOne behavior instead, comment out the maxPathKey property -->
  <managed-property>
    <property-name>maxPathKey</property-name>
    <value>TRAIN_DEMO_MAX_PATH_KEY</value>
  </managed-property>
</managed-bean>

```

How to Bind to the Train Model in JSF Pages

Each stop in the train corresponds to one JSF page. On each page, you use one train component and optionally a `trainButtonBar` component to provide buttons that allow the user to navigate through the train.

Before you begin:

It may help to understand how a train component's attributes affect functionality. See [Using Train Components to Create Navigation Items for a Multistep Process](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Navigation Components](#).

To bind the train component to the train model:

1. In the Components window, from the General Controls panel, in the Location group, drag and drop a **Train** onto the JSF page. Optionally drag and drop a **Train Button Bar**.
2. Bind the component. If your `MenuModel` implementation for a train model returns a `rowData` object similar to the public abstract class `oracle.adf.view.rich.model.TrainStopModel`, you can use the simplified form of train binding in the train components, as shown in [Example 20-15](#).

Example 20-15 Simple Implementation of a Train Component Bound to a Menu Model

```
<af:train value="#{trainMenuModel}"/>
<af:trainButtonBar value="#{trainMenuModel}"/>
```

Example 20-16 commandNavigationItem component Bound to a Train Stop

```
<af:train value="#{aTrainMenuModel}" var="stop">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem
      text="#{stop.label}"
      action="#{stop.outcome}"
      ...
    </af:commandNavigationItem>
  </f:facet>
</af:train>
```

The `trainMenuModel` EL expression is the managed bean name for the train model (see [Example 20-14](#)).

If you cannot use the simplified binding, you must bind the `train` value to the train model bean, manually add the `nodeStamp` facet to the train, and to that, add a `commandNavigationItem` component, as shown in [Example 20-16](#).

Determining Components at Runtime

This chapter describes how to use the ADF Faces `dynamicComponent` with forms and tables, and how to create the `AttributesModel` to support it. If your application uses the full Fusion technology stack, then your model is created for you, and you can use data controls to create the dynamic component. See *Creating a Basic Databound Page* and *Creating ADF Databound Tables* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About Determining Components at Runtime](#)
- [Creating the Model for a Dynamic Component](#)
- [Adding a Dynamic Component as a Form to a Page](#)
- [Adding a Dynamic Component as a Table to a Page](#)
- [Using Validation and Conversion with Dynamic Components](#)
- [Using Dynamic and Static Components Together](#)

About Determining Components at Runtime

The ADF Faces dynamic component is a component that determines the components to display, and their values, at runtime. You can use this component when you are unsure of what components are needed at runtime.

There may be cases when you don't know the exact components needed at runtime. For example, your business objects may contain attributes that are only valid for certain instances, and you only want to display those attributes when necessary. Using standard components, you might need to incorporate expensive logic to determine whether or not to display certain fields.

Another example of needing a dynamic interface might be when multiple pages share the same data source. If the attributes on the object are likely to change, it would require a change to all the pages bound to it.

ADF Faces provides a dynamic component (`af:dynamicComponent`) that determines what components to display, and their values, at runtime. This component will only display the needed attributes for each rendered instance. Additionally, when you make changes to the associated business service, those changes will be reflected by the dynamic component, without any change needed to the UI code.

You can use the dynamic component in either a form or a table. At runtime, the needed components will be rendered within the form or table, in place of the dynamic component. The component that is rendered is based on the attribute's data type. For example, if the attribute is a `String`, then an `inputText` component is used. If it is a `Date`, then the `inputDate` component is used. Following are the components supported by the dynamic component:

- `inputText`

- `inputDate`
- `inputListOfValues`
- `selectOneChoice`
- `selectManyChoice`
- `selectOneListbox`
- `selectManyListbox`
- `selectOneRadio`
- `selectBooleanRadio`
- `selectBooleanCheckbox`
- `selectManyCheckbox`

For a form, the dynamic component is wrapped in an iterator component. Like the collection-based components, the iterator is bound to the complete collection, in this case, a collection of attributes in the `AttributesModel` object. The iterator stamps through each instance on this model, copying the data for the current instance into its `var` attribute. The dynamic component then accesses that `var` value for each instance using its `attributeModel` attribute, and uses that information to determine the type of component to use, how to configure it, and its value.

For example, say you want to create a form that displays employee data, as shown in [Figure 21-1](#).

Figure 21-1 Form Created with a Dynamic Component

The screenshot shows a web form with the following fields and values:

- Employee Number: 1234
- Department Number: 10
- Name: SMITH
- Manager: (empty)
- Salary: 150000
- Hier Date: Mon Jul 12 00
- Department Name: RESEARCH

At the bottom of the form are two buttons: "Previous" and "Next".

You can create an `AttributesModel` that contains information about each of the attributes to display, as well as a way to access the values for each instance, and then create a form using just the dynamic component, as shown in the following example.

```
<af:panelFormLayout id="pf1">
  <af:iterator value="#{TestDynCompBean.attributesModel.attributes}"
    var="attr" id="dyit1">
    <af:dynamicComponent value="#{TestDynCompBean.value[attr.name]}" id="dyipt3"
      attributeModel="#{attr}" />
  </af:iterator>
</af:panelFormLayout>
```

For static table, each column and the component inside that column, is statically defined in the page at design time. For a dynamic table, both the number of columns and component inside that column are dynamically defined at runtime, using the following components:

- `af:table`: Defines a table, including how to get the value, using the `var` attribute (see [Using Tables, Trees, and Other Collection-Based Components](#)).
- `af:iterator`: Defines the collection of attributes. A column, and a component in each column, will be built for each attribute.
- `af:column`: Defines a column that will be stamped once for each attribute. If there are 10 attributes in the iterator, then there will be 10 columns, all stamped from this same column definition.
- `af:dynamicComponent`: Defines the component for that column. The type, value, and so on, are obtained from that attribute.

The following example shows a dynamic component used in a table.

```
<af:table value="#{TestDynCompBean.values}" var="row"
    varStatus="vs" rowSelection="single"
    id="t1" width="100%">
  <af:iterator value="#{TestDynCompBean.attributesModel.attributes}" id="itr1"
    var="col">
    <af:column headerText="#{col.label}" id="c1">
      <af:dynamicComponent value="#{row[col.name]}"
        attributeModel="#{col}" id="dc1"/>
    </af:column>
  </af:iterator>
</af:table>
```

You can also use the dynamic component to create groupings of attributes. For example, say you are using a dynamic component to create a form that displays attributes for an employee. You want the employee information (such as first name, last name, salary, etc.) to be in one group, and you want the department information (such as department name, department number, etc.) in another group. You might create a category named "Employee Info" and another category named "Department Info." In the `AttributesModel`, you assign some attributes on the Employee object to be in the categories. These categories are held in the `hierarchicalAttributes` property of the `AttributesModel` object.

To create the groups on the page, you use a switcher component with two facets. The first facet will handle all attributes that belong to a category and the second will handle the "flat" group (that is, attributes that do not belong to a category).

When using groups, instead of being bound to `attributes` property in the `AttributesModel`, the main iterator is bound to `hierarchicalAttributes` property, which defines the root level attributes. As the iterator iterates over the collection, those category values are held in the variable named `attr`. In the first group, the iterator is bound to the variable `attr`, and so iterates through those, holding the value of the descriptors (the list of child attributes belonging in that category) in the variable `nestedAttr`. The child dynamic component then accesses this variable to determine the type of component and value to display for each record, as shown in the following example

```
<af:panelFormLayout id="pf1">
  <af:iterator value="#{TestDynCompBean.attributesModel.hierarchicalAttributes}"
    var="attr" id="dyit1">
    <af:switcher id="sw" facetName="#{attr.descriptorType}"
      defaultFacet="ATTRIBUTE">
      <f:facet name="GROUP">
        <af:group id="gg" title="#{attr.label}">
          <af:outputText value="#{attr.label}" id="ot2"/>
          <af:iterator id="it2" value="#{attr.descriptors}" var="nestedAttr">
```

```

        <af:dynamicComponent
            value="#{TestDynCompBean.value[nestedAttr.name]}" id="ndync1"
            attributeModel="#{nestedAttr}" />
    </af:iterator>
</af:group>
</f:facet>
<f:facet name="ATTRIBUTE">
    <af:dynamicComponent value="#{TestDynCompBean.value[attr.name]}"
        id="iinerit1" attributeModel="#{attr}" />
</f:facet>
</af:switcher>
</af:iterator>
</af:panelFormLayout>

```

The main iterator iterates through the root level attributes in the `attributeModel`. For any root attributes that do not belong to a group (i.e. an `Attribute`-typed root attribute), a dynamic component is created for it in the `ATTRIBUTE` facet. For any root attributes that do have attributes under it (i.e. a `Group`-type root attribute), the attributes are added to the `GROUP` facet. In that facet, another iterator iterates through each regular attribute in that group, and creates a dynamic component based on each regular attribute.

Creating the Model for a Dynamic Component

ADF Faces provides the `AttributeModel` class that has a collection of attributes. Each of these attributes is described by a `BaseAttributeDescriptor` object. You can create a model with or without groups for a dynamic component. For a model without groups, you must create `BaseAttributeDescriptor` object, `AttributesModel` object, and a managed bean. For a model with groups, you must create a `GroupAttributeDescriptor` object.

The `AttributesModel` class is a collection of attributes, each attribute described by a `BaseAttributeDescriptor` object. This object provides the metadata used by the dynamic component to determine how to display the data, including the component type, the name, label, and description. You will need to extend the `BaseAttributeDescriptor` class and the `AttributesModel` class to provide the needed information for your dynamic components.

If you want to use groups with your dynamic component, then you must also create a `GroupAttributeDescriptor` object.

How to Create the Model Without Groups

To create the model, you need to create the `BaseAttributeDescriptor` object, the `AttributesModel` object, and a managed bean for the page that the dynamic component and iterator can use to access the data and metadata for the attributes.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To create the model:

1. Create an attribute definition class that describes each property on an attribute. It is this metadata that the dynamic component will use to determine the component

to use to display the data, and how to configure it. For example, this definition class might contain getter methods for the following properties on an attribute:

- name
- label
- dataType

For an example of an attribute definition class, see the `TestAttributeDef` inner class on the `TestDynamicComponentPageDef` class in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the Application Sources directory of the ADF Faces application.

2. Create another class that defines each attribute, using the definition class created in Step 1. This class should define each attribute, and then return a list of attributes with that metadata populated.

For example, an `Employee` object might have the following attributes:

- Ename
- Empno
- Deptname
- Deptno
- Manager

The following example shows how you might create an attribute definition, given the metadata created in Step 1.

```
public void addAttributeDef (String name, String label, Class dataType) {
    TestAttributeDef attributeDef = new TestAttributeDef(name, label, dataType);
    _attributes.put(name, attributeDef);
}
```

The following example then shows how to return a list of `Employee` object attributes with the attribute definitions populated.

```
public void setupAttributes()
{
    _attributes = new HashMap<String, TestAttributeDef>();
    addAttributeDef("Ename", "Employee Name", null, "Name", 10, 20,
String.class);
    addAttributeDef("Empno", "Employee Number", null,
"Employee Number", 10, 20, Number.class);
    addAttributeDef("Deptname", "Department Name", null, "Department Name", 10,
20, String.class);
    addAttributeDef("Deptno", "Department Number", null, "Department Number", 10,
20, Number.class);
    addAttributeDef("Manager", "Manager", null, "Manager", 10, 20, Number.class);
}
```

For an example of a complete attribute definition class, see the `TestDynamicComponentPageDef` class in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the Application Sources directory of the ADF Faces application.

3. Create a class for the `BaseAttributeDescriptor` object. This class must extend the `BaseAttributeDescriptor` class, and needs to access the metadata properties for each attribute. The following example shows how the `Name` and `DataType` might be returned.

```

public class TestAttributeDescriptor extends BaseAttributeDescriptor {
    public TestAttributeDescriptor(TestAttributeDef attributeDef) {
        _attributeDef = attributeDef;
    }

    public Object getId() {
        return getName();
    }

    public String getName() {
        return _attributeDef.getName();
    }

    public Class getDataType() {
        return _attributeDef.getDataType();
    }
}

```

For a complete example, see the `TestAttributeDescriptor` inner class of the `TestDynamicComponentBean` managed bean in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the `Application Sources` directory of the ADF Faces application.

4. Create a class that extends the `AttributesModel` class. This class needs to create the attributes from the `BaseAttributeDescriptor` object as a list.

```

public class TestAttributesModel extends AttributesModel {
    public TestAttributesModel() {
        _flatAttributes = new ArrayList<BaseAttributeDescriptor>();
    }

    public List<BaseAttributeDescriptor> getAttributes() {
        return _flatAttributes;
    }

    private void _setupAttributesFromDefinition() {
        Map<String, List<BaseAttributeDescriptor>>();
        List<TestDynamicComponentPageDef.TestAttributeDef> attributeList =
        _pageDef.getAttributeDefs();
        . . .
        for (TestDynamicComponentPageDef.TestAttributeDef demoAttrDef :
        attributeList) {
            TestAttributeDescriptor attrDesc = new
            TestAttributeDescriptor(demoAttrDef);
            _flatAttributes.add(attrDesc);
        }
    }
}

```

For a complete example, see the `TestAttributesModel` inner class of the `TestDynamicComponentBean` managed bean in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the `Application Sources` directory of the ADF Faces application.

5. Create the managed bean. The managed bean needs to return the populated `AttributesModel` objects for the page, as well as provide any needed logic for the page. For example, if you want to use the dynamic component in a form, and you want the user to be able to scroll to previous and next records, you need to provide that logic. The following example shows the code for returning the `AttributesModel`, as well as the logic for accessing next and previous records using Action events.

```
public class TestDynamicComponentBean {
    public TestDynamicComponentBean() {
        _attrsModel = new TestAttributesModel();
    }

    public AttributesModel getAttributesModel() {
        return _attrsModel;
    }

    public Map[] getValues() {
        return _DATA;
    }
    public Map getValue() {
        return _DATA[currentRowIndex];
    }

    public void next(ActionEvent actionEvent) {
        if (currentRowIndex < _DATA.length - 1)
            currentRowIndex++;
    }

    public boolean getNextEnabled() {
        return (currentRowIndex < (_DATA.length - 1));
    }

    public void previous(ActionEvent actionEvent) {
        if (currentRowIndex > 0)
            currentRowIndex--;
    }

    public boolean getPreviousEnabled() {
        return currentRowIndex > 0;
    }
}
```

For a complete example (including how the managed bean sets up the data for `AttributesModel`), see the `TestDynamicComponentBean` managed bean in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the `Application Sources` directory of the ADF Faces application.

How to Create the Model Using Groups

If you want to use groups with your dynamic component, then you must also create a `GroupAttributeDescriptor` object, which holds the metadata required for the group information, including the list of attributes that belong to it. You will also need to supply the logic needed by the iterator and dynamic component to display the groups.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To create the model using groups:

1. Create the attribute definition class, as described in Step 1 of [How to Create the Model Without Groups](#), but include an attribute definition for `category` as a `String`. The `category` will be used to define the groups.
2. When you create your attributes, assign values for `category` for the attributes that need them. For example, you might assign the value "Employee Personal" to the

Empname and Empno attributes, and the value "Department Info" to the Deptno and Deptname attributes.

3. Include category in the BaseAttributeDescriptor class by providing a get method for it. However, the get method should have the signature `getGroupName`. The following example shows how the get method might be coded.

```
public String getGroupName() {
    return _attributeDef.getCategory();
}
```

4. Create a class for the GroupAttributeDescriptor object that extends GroupAttributeDescriptor. This object provides a name for the group and a list of the attributes in the group. The following example shows a GroupAttributeDescriptor class.

```
public class TestGroupAttributeDescriptor extends GroupAttributeDescriptor {
    public TestGroupAttributeDescriptor(String groupName,
    List<BaseAttributeDescriptor> attributeList) {
        _groupName = groupName;
        _flatAttributes = attributeList;
    }

    @Override
    public String getName() {
        return _groupName;
    }

    public List<? extends Descriptor> getDescriptors() {
        return _flatAttributes;
    }

    public String getDescription() {
        return "This is a group";
    }

    public String getLabel() {
        return _groupName;
    }

    private List<BaseAttributeDescriptor> _flatAttributes;
    private String _groupName;
}
```

5. In your AttributesModel class, add the hierarchicalAttributes that represent the groups. To do this, you might add logic that examines the BaseAttributeDescriptor object, and if it contains a group name, it adds that to the hierarchicalAttributes object. The following example shows how you might code that logic.

```
for (TestDynamicComponentPageDef.TestAttributeDef demoAttrDef : attributeList) {
    TestAttributeDescriptor attrDesc = new TestAttributeDescriptor(demoAttrDef);
    _flatAttributes.add(attrDesc);
    String groupName = attrDesc.getGroupName();
    if (groupName != null && !groupName.isEmpty()) {
        List<BaseAttributeDescriptor> list = groupMap.get(groupName);
        if (list == null) {
            list = new ArrayList<BaseAttributeDescriptor>();
            groupMap.put(groupName, list);
        }
        list.add(attrDesc);
    }
}
```

```
        } else {  
            _hierAttributes.add(attrDesc);  
        }  
    }  
  
    for (String groupName : groupMap.keySet()) {  
        TestGroupAttributeDescriptor groupMetadata =  
            new TestGroupAttributeDescriptor(groupName, groupMap.get(groupName));  
        _hierAttributes.add(groupMetadata);  
    }  
}
```

For a complete example, see the `TestAttributesModel` inner class of the `TestDynamicComponentBean` managed bean in the `oracle.adfdemo.view.feature.rich.dynamicFaces` package, found in the Application Sources directory of the ADF Faces application.

Adding a Dynamic Component as a Form to a Page

ADF Faces `panelFormLayout` component is an iterator that access the `AttributeModel` to get the attributes and their definitions. You must use this iterator when you want to use a dynamic component in a form without grouping attributes. If you want to group the attributes on the form while using the dynamic component then you use a switcher component to separate the grouped and ungrouped attributes.

When you use the dynamic component in a form, you need to include an iterator. The iterator is what access the `AttributesModel` to get the attributes and their definitions. The dynamic component then gets the information for each instance the iterator stamps out, and determines what component to use, and how to configure it.

If you want to group your attributes in the form, then you also need to use a switcher component with two facets. One facet will display the groups with their associated attributes, while the other facet will display any attributes not associated with a group.

How to Add a Dynamic Component as a Form without Groups to a Page

To use a dynamic component in a form, you use the `panelFormLayout` component, an iterator, and the dynamic component. If you want to provide a way to navigate between records in the form, then you also need to use buttons.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To add a dynamic component as a form without groups to a page:

1. Create a `panelFormLayout` component, as described in [Arranging Content in Forms](#).
2. In the Components window, from the Operations panel, drag and drop an **Iterator** as a child to the `panelFormLayout` component.
3. In the Properties Window, set the following:

- **Value:** An EL expression that resolves to the `attributes` property on the `AttributesModel` object, for example:


```
#{TestDynCompBean.attributesModel.attributes}"
```
 - **Var:** A `String` that can be used to access the attributes on the model, for example: `attr`
4. From the Components window, drag and drop a **Dynamic Component** as a child to the `iterator` component.
 5. In the Properties Window, set the following:
 - **AttributeModel:** An EL expression that resolves to the variable created for the iterator, for example: `#{attr}`.
 - **Value:** An EL expression that resolves to each attribute name on the `AttributesModel` object, for example:


```
#{TestDynCompBean.value[attr.name]}"
```
 6. If you want to provide logic to navigate between the records, add the buttons in a `panelGroupLayout` component, and use the action event to access the next and previous records. For information about using buttons, see [How to Use Buttons and Links for Navigation and Deliver ActionEvents](#).

The following example shows buttons whose `actionListener` attributes are bound to logic on a managed bean that navigates between the records in the model.

```
<af:panelGroupLayout layout="horizontal" id="pgl2">
  <af:button text="Previous" id="cb2"
  actionListener="#{TestDynCompBean.previous}"
  disabled="#{!TestDynCompBean.previousEnabled}"/>
  <af:button text="Next" id="cb1" actionListener="#{TestDynCompBean.next}"
  disabled="#{!TestDynCompBean.nextEnabled}"/>
</af:panelGroupLayout>
```

The following example shows the corresponding managed bean code.

```
public Map[] getValues() {
    return _DATA;
}

public Map getValue() {
    return _DATA[currentRowIndex];
}

public void next(ActionEvent actionEvent) {
    if (currentRowIndex < _DATA.length - 1)
        currentRowIndex++;
}

public boolean getNextEnabled() {
    return (currentRowIndex < (_DATA.length - 1));
}
```

How to Add a Dynamic Component as a Form with Groups to a Page

When you want to group attributes on form using a dynamic component, you need to place the attributes that are part of a group in one part of the form, and the attributes that don't belong to a group, in another. You use the facets of a `switcher` component to separate the two.

The switcher component is a child of an iterator component. But instead of being bound to the attributes of the `AttributesModel`, this iterator is bound to the `hierarchicalAttributes` of the model, and stores those objects in its variable. Another iterator, in the facet of the switcher, is then bound to the descriptors included in the `hierarchicalAttributes` object. It is from this iterator that the dynamic component gets its information.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To add a dynamic component as a form with groups to a page:

1. Create a `panelFormLayout` component, as described in [Arranging Content in Forms](#).
2. In the Components window, from the Operations panel, drag and drop an **Iterator** as a child to the `panelFormLayout` component.
3. In the Properties window, set the following:
 - **Value:** An EL expression that resolves to the `hierarchicalAttributes` property on the `AttributesModel` object, for example:

```
#{TestDynCompBean.attributesModel.hierarchicalAttributes}"
```
 - **Var:** A `String` that can be used to access the attributes on the model, for example: `attr`
4. Drag and drop a **Switcher** as a child to the `iterator` component.
5. In the Properties window, set the following:
 - **FacetName:** Enter an EL expression that resolves to the `descriptorType` property on the values returned by the variable on the parent iterator. For example:

```
#{attr.descriptorType}
```
 - **DefaultFacet:** Name the facet that will hold attributes not returned by `#{attr.descriptorType}` or that do not match any of the defined facets, for example, `ATTRIBUTE`.
6. From the Structure window, right-click the switcher component and choose **Insert Inside Switcher > Facet**.
7. In the Insert Facet dialog, name the facet `GROUP`. This will create the facet that will contain the attributes that belong to a group.
8. In the Components window, from the Text and Selection panel, drag and drop an **Output Text**. This component will display the group name.
9. In the Properties window, bind the **Value** to the `label` property of the returned groups, using the parent iterator variable. For example, `#{attr.label}`.
10. Drag and drop an **Iterator** as a child to the `outputText` component. This iterator will iterate through the attributes in each group, and is what the dynamic component will use to display the components and values.
11. In the Properties window, set the following:
 - **Value:** An EL expression that resolves to the Descriptor objects returned by the first iterator, using its variable, for example, `#{attr.descriptors}`.

- **Var:** A `String` that can be used to access the attributes in the descriptors, for example: `nestedAttr`
12. Drag and drop a **Dynamic Component** as a child to the second iterator.
 13. In the Properties Window, set the following
 - **AttributeModel:** An EL expression that resolves to the variable created for the second iterator, for example: `#{nestedAttr}`.
 - **Value:** An EL expression that resolves to each attribute name returned by the second iterator, for example:


```
#{TestDynCompBean.value[nestedAttr.name]}
```
 14. In the Structure window, select the `outputText` and `iterator` components, right-click these selections, and choose **Surround With**. In the Surround With dialog, select **Group** and click **OK**.
In the Properties window, bind **Title** to the label property of the groups returned by the first iterator, for example, `#{attr.label}`.
 15. Drag and drop another Facet as a child to the switcher component. Name this facet `ATTRIBUTE`.
 16. Drag and drop another **Dynamic Component** as a child to the `ATTRIBUTE` facet, and set the following:
 - **AttributeModel:** An EL expression that resolves to the variable created for the main iterator, for example: `#{attr}`.
 - **Value:** An EL expression that resolves to each attribute name returned by the main iterator, for example:


```
#{TestDynCompBean.value[attr.name]}
```

Adding a Dynamic Component as a Table to a Page

ADF Faces allows you to use the dynamic component in a table by wrapping the table columns in an iterator. The iterator gets the attributes and their definition from the `AttributeModel`. The table column then uses this information to determine its header text. You can also group your attributes by using a switcher component with two facets.

When you use the dynamic component in a table, you need the table's columns to be wrapped in an iterator. The iterator is what access the `AttributesModel` to get the attributes and their definitions. The column uses this information to determine its header text. The dynamic component then gets the information for each attribute instance the iterator stamps out, and determines what component to use, and how to configure it, and at the same time, uses the variable on the table to determine the data to display.

If you want to group your attributes in the table, then you also need to use a switcher component with two facets. One facet will display the groups with their associated attributes, while the other facet will display any attributes not associated with a group.

How to Add a Dynamic Component as a Table Without Groups to a Page

To use a dynamic component in a table, you bind the table component to the data. Instead of a `column` component, the direct child of the table is an iterator bound to the `attributes` property of the `AttributesModel`. The column is a child of the iterator. The dynamic component is a child to the column.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To add a dynamic component as a table without groups to a page:

1. In the Components window, from the Data Views panel, drag and drop a **Table** to open the Create ADF Faces Table wizard.

For information about configuring tables, see [Displaying Data in Tables](#).

2. Select **Bind Data Now**, and use the **Browse** button to choose the model that holds the table's data.
3. Select **Generate columns dynamically at runtime**.

Note:

This option only displays once you've chosen a model for the table.

4. In the **Attributes Collection** field, enter an n EL expression that resolves to the `attributes` property on the `AttributesModel` object you created in [How to Create the Model Without Groups](#), for example:

```
#{TestDynCompBean.attributesModel.attributes}"
```

The following example shows the declarative code created for a dynamic table.

```
<af:table value="#{TestDynCompBean}" var="row" rowBandingInterval="0" id="t1">
  <af:iterator value="#{TestDynCompBean.attributesModel.attributes}"
    var="column" id="i1">
    <af:column headerText="#{column.label}" id="c1">
      <af:dynamicComponent attributeModel="#{column}" value="#{row[column.name]}"
        id="dc1"/>
    </af:column>
  </af:iterator>
</af:table>
```

How to Add a Dynamic Component as a Table with Groups to a Page

When you group attributes in a table using a dynamic component, the groups display as parent columns. [Figure 21-2](#) shows the `Employee` object with certain attributes included in the Employee Personal and Department Info groups.

Figure 21-2 Table with Dynamic Component that Uses Groups

Manager	Hier Date	Employee Personal			Department Info	
		Employee Number	Name	Salary	Department Number	Departme
	Mon Jul 12 00	1234	SMITH	150000	10	RESEARC
1234	Tue Apr 25 00	2345	SCOTT	120000	10	RESEARC
	Tue May 11 01	5678	THOMAS	90000	20	SALES
	Mon Jul 12 00	3234	Mike	150000	10	RESEARC
1234	Tue Apr 25 00	2345	Tom	120000	10	RESEARC

You use a switcher component to separate the attributes that belong to groups from the ones that do not. But instead of being bound to the attributes of the `AttributesModel`, this main iterator is bound to the `hierarchicalAttributes` of the model, and stores those objects in its variable. Another iterator, in the facet of the switcher, is then bound to the descriptors included in the `hierarchicalAttributes` object. It is from this iterator that the dynamic component gets its information and the component can display the groups of attributes.

The `hierarchicalAttributes` of the `AttributeModel` defines the root-level attributes of the `AttributeModel`. If a root level attribute is a normal attribute, meaning the attribute type of `ATTRIBUTE`, then this attribute does not belong to any group, and the table will render one column for the attribute. If a root level attribute is a grouped attribute, meaning the attribute type is `GROUP`, then there will be a list of regular attributes that belong to this group. The iterator in the facet will iterate through this group, and each attribute inside that group will be stamped as a column. For example, `Employee Personal` is a `GROUP`-type attribute, and it contains 3 regular attributes: `Employee Number`, `Name` and `Salary`, as shown in [Figure 21-2](#).

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To add a dynamic component as a table with groups to a page:

1. In the Components window, from the Data Views panel, drag and drop a **Table** to open the Create ADF Faces Table wizard.
For information about configuring tables, see [Displaying Data in Tables](#).
2. Select **Bind Data Now**, and use the **Browse** button to choose the model that holds the table's data.
3. Select **Generate columns dynamically at runtime**.

 **Note:**

This option only displays once you've chosen a model for the table.

4. Select **Include Column Groups**.
5. In the **Attributes Collection** field, enter an n EL expression that resolves to the `attributes` property on the `hierarchicalAttributes` property on the `AttributesModel` object you created in [How to Create the Model Without Groups](#), for example:

```
#{TestDynCompBean.attributesModel.heirarchicalAttributes}"
```

The following example shows the declarative code created for a dynamic table with groups.

```
<af:table value="#{TestDynCompBean}" var="row" rowBandingInterval="0" id="t2">
  <af:iterator value="#{TestDynCompBean.attributesModel.heirarchicalAttributes}"
    var="column" id="i2">
    <af:column headerText="#{column.label}" id="c2">
      <af:switcher defaultFacet="ATTRIBUTE"
        facetName="#{column.descriptorType}" id="s1">
        <f:facet name="GROUP">
          <af:iterator value="#{column.descriptors}" var="nestedCol" id="i3">
            <af:column headerText="#{nestedCol.label}" id="c3">
              <af:dynamicComponent attributeModel="#{nestedCol}"
                value="#{row[nestedCol.name]}" id="dc2"/>
            </af:column>
          </af:iterator>
        </f:facet>
        <f:facet name="ATTRIBUTE">
          <af:dynamicComponent attributeModel="#{column}"
            value="#{row[column.name]}" id="dc3"/>
        </f:facet>
      </af:switcher>
    </af:column>
  </af:iterator>
</af:table>
```

Using Validation and Conversion with Dynamic Components

ADF Faces allows you to convert and validate the data submitted by users to the dynamic components. You must add the needed converter or validator tag to the dynamic components and specify the attribute to validate or convert while building your application.

You add conversion and validation to dynamic components by adding the needed converter or validator tag, and specifying the attribute to validate or convert. Instead of the `enabled` attribute, dynamic components use the `disabled` attribute.

Before you begin:

It may be helpful to have an understanding of how dynamic components determine what they should display. See [About Determining Components at Runtime](#).

To use a validator or converter with a dynamic component:

1. In the Components window, from the Operations panel, drag and drop the needed converter or validator as a child to the dynamic component.
2. In the Properties window, in the Other section, click the icon that appears when you hover over the **Disabled** field and choose **Expression Builder**.
3. In the Expression Builder enter an expression that resolves to the attribute and provides the needed pattern. The following example shows two converters and validators that might be used for the different attributes represented by the dynamic component. The `DateTime` converter will be run on the `Hiredate` attribute, the `Number` converter will be run on the `Sal` attribute, the `Length` validator will be run on the `Job` attribute and the `LongRange` validator will be run on the `Sal` attribute.

```
<af:dynamicComponent value="#{DynCompBean.value[attr.name]}"
  attributeModel="#{attr}" id="dc1"/>
```

```

<af:convertDateTime disabled="#{attr.name == 'Hiredate' ? false : true}"
    pattern="yyyy/MM/dd"/>
<af:convertNumber disabled="#{attr.name == 'Sal' ? false : true}"
    pattern="#,###,###" />
<af:validateLength disabled="#{attr.name == 'Job' ? false : true}"
    maximum="10" hintMaximum="maximum length is 10"/>
<af:validateLongRange disabled="#{attr.name == 'Sal' ? false : true}"
    minimum="1000"/>
</af:dynamicComponent>

```

Using Dynamic and Static Components Together

ADF Faces allows you to place a static component before and after a dynamic form. You can also create complex forms using dynamic and static components within the form.

When you create a dynamic form, you essentially create a block of components that are rendered dynamically. It is possible place static components before and after that block of components, but you can not place static components within that block.

Even so, you can create complex forms with a number of different dynamic and static components. And you can have static components interspersed with dynamic components at a fairly granular level.

For example, say you have a form where you need to display name and address information, both in English and in Japanese. For the interest of this example, let's say the Japanese information will be displayed using dynamic components, while the English will be displayed with static components. [Figure 21-3](#) shows how you might organize the fields by having the dynamic Japanese content display after the static content.

Figure 21-3 Dynamic Components Can Be Placed After Static Components

Name	
First Name	Scott
Middle Name	Aidan
Last Name	Smith
Japanese First Name	Daichi
Japanese Last Name	Suzuki
Address	
Street Address	123 Main Street
City	San Francisco
State	CA
Japanese Street Address	456 Capital Street
Japanese City	Tokyo
Japanese Province	Tokyo

The following example shows how you might create the form using static `inputText` components and dynamic components that access the attributes in the `Name` and `Address` groups.

```

<panelFormLayout ... >
  <af:group title="Name">
    <af:inputText value="#{myBean.firstName} id="it1" label="First Name"/>
    <af:inputText value="#{myBean.middleName} id="it2" label="Middle Name"/>
    <af:inputText value="#{myBean.lastName} id="it3" label="Last Name"/>
  </af:group>

```

```

<af:iterator
    value="#{DynCompBean.attributesModel.hierarchicalAttributes("Name")}"
    var="attr" id="iter1">
    <af:dynamicComponent id="dcl"
        value="#{DynCompBean.value[attr.name]}"
        attributeModel="#{attr}"/>
    </af:iterator>
</af:group>
<af:group title="Address">
    <af:inputText value="#{myBean.streetAddress}" id="it4"
        label="Street Address"/>
    <af:inputText value="#{myBean.City.inputValue}" id="it5" label="City"/>
    <af:inputText value="#{myBean.State.inputValue}" id="it6" label="State"/>
    <af:iterator
        value="#{DynCompBean.attributesModel.hierarchicalAttributes("Address")}"
        var="attr" id="iter1">
        <af:dynamicComponent id="dcl"
            value="#{DynCompBean[attr.name].inputValue}"
            attributeModel="#{attr}"/>
        </af:iterator>
    </af:group>
</panelFormLayout>

```

Now say that instead of having the Japanese name and address information separate from the English, you want them interspersed, as shown in [Figure 21-4](#).

Figure 21-4 Dynamic Components Can Be Placed With Static Components

The screenshot shows a web form with two main sections: "Name" and "Address".

Name Section:

- First Name:
- Japanese First Name:
- Middle Name:
- Last Name:
- Japanese Last Name:

Address Section:

- Street Address:
- Japanese Street Address:
- City:
- Japanese City:
- State:
- Japanese Province:

In this case, you would not use groups in the dynamic component, but instead would access only the needed attribute (and not all attributes), as shown in the following example.

```

<af:group title="Name">
    <af:inputText value="#{myBean.firstName}" id="it1" label="First Name"/>
    <af:iterator value="#{DynCompBean.attributesModel.attributes("Japanese First
Name")}"
        var="attr" id="iter1">
        <af:dynamicComponent id="dcl" value="#{DynCompBean.value[attr.name]}"
            attributeModel="#{attr}"/>
        </af:iterator>
    <af:inputText value="#{myBean.middleName}" id="it2" label="Middle Name"/>
    <af:inputText value="#{myBean.lastName}" id="it3" label="Last Name"/>
    <af:iterator value="#{DynCompBean.attributesModel.attributes("Japanese Last
Name")}"

```

```
        var="attr" id="iter1">
    <af:dynamicComponent id="dc1" value="{DynCompBean.value[attr.name]}"
        attributeModel="{attr}"/>
    </af:iterator>
</af:group>
<af:group title="Address">
    <af:inputText value="{myBean.streetAddress}" id="it4" label="Street Address"/>
    <af:iterator value="{DynCompBean.attributesModel.attributes('Japanese Street
Address')}">
        var="attr" id="iter1">
    <af:dynamicComponent id="dc1" value="{DynCompBean.value[attr.name]}"
        attributeModel="{attr}"/>
    </af:iterator>
    . . .
</af:group>
```

Part V

Using ADF Data Visualization Components

Part V documents the different ADF Faces data visualization components, including charts, gauges, NBoxes, pivot tables, Gantt charts, timelines, geographic and thematic maps, hierarchy viewers, treemaps, sunburst, and diagram components.

Specifically, it contains the following chapters:

- [Introduction to ADF Data Visualization Components](#)
- [Using Chart Components](#)
- [Using Picto Chart Components](#)
- [Using Gauge Components](#)
- [Using NBox Components](#)
- [Using Pivot Table Components](#)
- [Using Gantt Chart Components](#)
- [Using Timeline Components](#)
- [Using Map Components](#)
- [Using Hierarchy Viewer Components](#)
- [Using Treemap and Sunburst Components](#)
- [Using Diagram Components](#)
- [Using Tag Cloud Components](#)

Introduction to ADF Data Visualization Components

This chapter describes the ADF Data Visualization components, an expressive set of interactive ADF Faces components. The functionality shared across the components and with other ADF Faces components is also highlighted. The remaining chapters in this part of the guide provide detailed information about how to create and customize each component.

This chapter includes the following sections:

- [About ADF Data Visualization Components](#)
- [Common Functionality in Data Visualization Components](#)
- [Providing Data for ADF Data Visualization Components](#)

About ADF Data Visualization Components

The ADF Data Visualization components provide significant graphical and tabular capabilities for displaying and analyzing data in an efficient manner with reduced effort.

These components provide the following common features:

- They are full ADF Faces components that support the use of ADF data controls.
- They provide for declarative design time creation using the Data Controls Panel, the JSF visual editor, Properties window, and Components window.

Data visualization components include: charts, picto charts, gauges, NBoxes, pivot tables and pivot filter bars, geographic maps, thematic maps, Gantt charts, timelines, hierarchy viewers, treemaps, sunbursts, diagrams, and tag clouds.

The prefix `dvt:` occurs at the beginning of each component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Chart Component Use Cases and Examples

The chart components include ten types of charts with one or more variations for a total of over twenty different charts that you can use to display data. Chart components give you the capability of producing a variety of data visualizations that let you evaluate multiple data points on multiple axes in many ways. For example, a number of charts assist you in the comparison of results from one group with the results from another group.

Chart categories include:

- **Area (`areaChart`):** Represents data as a filled-in area. Use area charts to show trends over time, such as sales for the last 12 months. Area charts require at least two groups of data along an axis. The axis is often labeled with increments of time such as months.

- **Bar** (`barChart`): Represents data as a series of vertical or horizontal bars. Use bar charts to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.
- **Bubble** (`bubbleChart`): Represents data by the location and size of round data markers (bubbles). Use bubble charts to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble chart to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a chart allows you to examine productivity relative to salary and experience.
- **Combination** (`comboChart`): Chart that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use combination charts to compare bars and lines, bars and areas, lines and areas, or all three combinations.
- **Funnel** (`funnelChart`): Displays data as a stack of slices in a conical section. Use funnel charts to represent steps in a process. Funnel charts require actual values and target values against a stage or time value. For example, use a funnel chart to represent progress in different stages of the software development life cycle.
- **Line** (`lineChart`): Represents data as a line, as a series of data points, or as data points that are connected by a line. Line charts require data for at least two points for each member in a group. For example, a line chart over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, and Asia. Use line charts to compare items over the same time.
- **Pie** (`pieChart`): Represents a set of data items as proportions of a total. The data items are displayed as sections of a circle causing the circle to look like a sliced pie. Use pie charts to show the relationship of parts to a whole such as how much revenue comes from each product line.
- **Scatter** (`scatterChart`): Represents data by the location of data markers. Use scatter charts to show correlation between two different kinds of data values such as sales and costs for top products. Use scatter charts in particular to see general relationships among a number of items.
- **Spark** (`sparkChart`): A simple, condensed chart that displays trends or variations in a single data value, typically stamped in the column of a table or in line with related text. Spark charts have basic conditional formatting. Since spark charts contain no labels, the adjacent columns of a table or surrounding text provide context for spark chart content.
- **Stock** (`stockChart`): Represents information related to trade stocks, specifically the opening, high, low, and closing prices of stocks. Use stock charts to explore trends in stock prices, fluctuations and volume of trade. Depending on the type of chart chosen, each stock marker displays two to four values, not counting the optional volume marker. When the volume of trading is included, the volume appears as bars in the lower part of the chart.

Figure 22-1 shows examples of area, bar, bubble, combination, funnel, line, pie, and scatter charts.

Figure 22-1 Example of ADF Data Visualization Charts



Figure 22-2 shows a line sparkchart displaying sales trends in a table column.

Figure 22-2 Sparkchart of Sales Trends

Stock Symbol	Prices for 2008
AAPL	
CSCO	
GOOG	
HPQ	
IBM	
INTC	
MSFT	
ORCL	

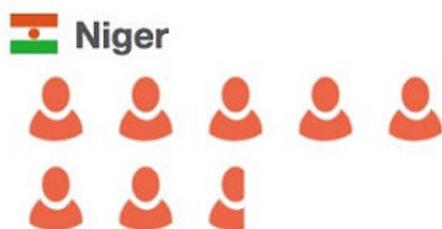
For more information including additional use cases and examples, see [Using Chart Components](#).

Picto Chart Use Cases and Examples

Picto Charts are a versatile tool that can be used along with other components to display data in a visually appealing manner. There are several use cases.

Picto charts can be used to enhance statements about absolute numbers and act as a visual aid. Figure 22-3 shows picto charts reflecting the absolute value mentioned in the accompanying statements.

Figure 22-3 Picto Charts with Absolute Values



United States

1.9 children per woman

South Korea

1.2 children per woman

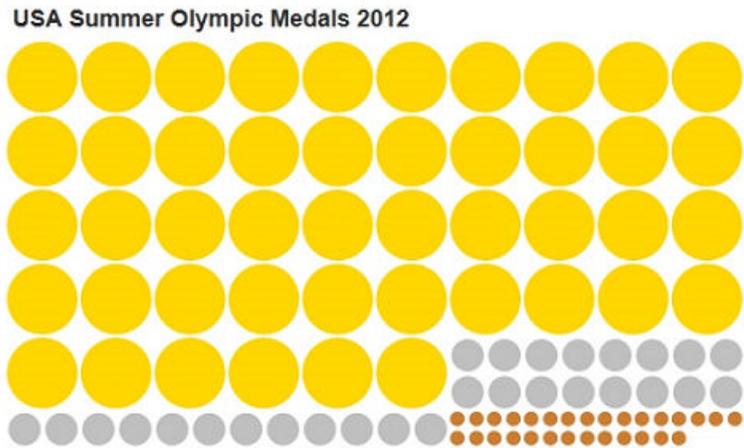
Picto Charts can be used to highlight a portion of a population as a comparison. This works best when the ratio is small. [Figure 22-4](#) shows a single picto chart configured to highlight three out of twenty human figures to illustrate the point made in the accompanying statement.

Figure 22-4 Simple Picto Chart to illustrate a Part of a Population



Another use case of picto charts is to create a comparison between parts of a population by highlighting the distribution of parts. For example, [Figure 22-5](#) shows three picto charts together in the default flow layout, each configured to show the number of medals won by the USA at the 2012 Summer Olympics. The three picto charts have been configured to correlate the size of the circle with the rank of the medal.

Figure 22-5 Picto Chart with varying sizes of items



Picto charts may also be used a set of singletons in order to display unique information for each data point. Figure 22-6 shows a collection of picto charts to represent USA's medal count in the 2014 Winter Olympics. The `count` attribute for each picto chart is set to 1 and the tooltip for each picto chart is configured to show the name of the athlete who won that medal.

Figure 22-6 A collection of Singleton Picto Charts with Unique Information



Multiple picto charts can be arranged within a table to highlight business data. Figure 22-7 shows a table with sales figures of Apple products in the years 2011 and 2012. Each square represents 10 million units.

Figure 22-7 Picto Charts arranged within Table

Product	2011	2012	Total (million units)
iPhone			132
iPad			17
iPod			77
Mac			35

Gauge Component Use Cases and Examples

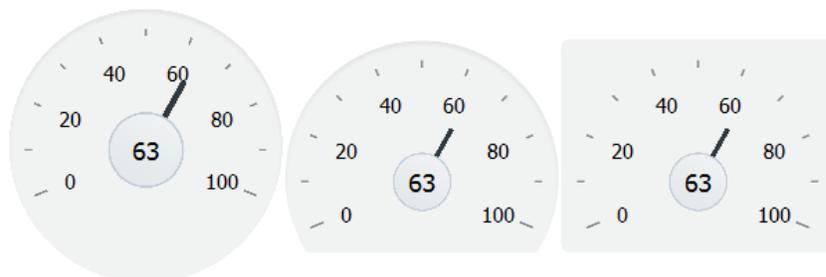
The gauge (*gauge*) component is a measuring instrument for indicating a quantity such as sales, stock levels, temperature, or speed. Gauges typically display a single data value, often more effectively than a chart. Using thresholds, gauges can show state information such as acceptable or unacceptable ranges using color.

The following kinds of gauges can be produced by this component:

- **Dial:** Displays a metric value plotted on a circular axis. The gauge's background attribute determines whether the gauge's background is displayed as a rectangle, circle, or semicircle. An indicator points to the dial gauge's metric value on the axis.

[Figure 22-8](#) shows three dial gauges with backgrounds set to full circle, partial circle, and rectangle. In all three examples, the gauge's metric value is 63.

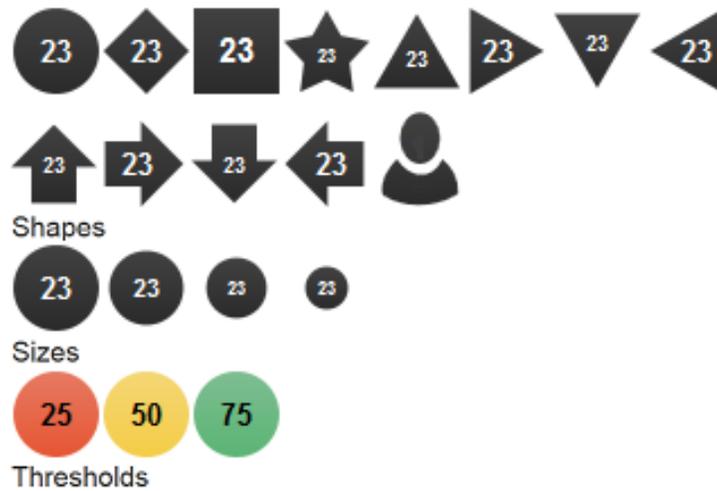
Figure 22-8 Dial Gauge Examples



- **LED (lighted electronic display):** Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of shapes are available for LED gauges, including round or rectangular shapes that use color to indicate status, and triangles or arrows that point up, left, right, or down in addition to a color indicator.

[Figure 22-9](#) shows LED gauges configured with a variety of shapes, sizes, and thresholds.

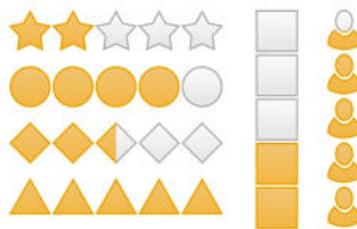
Figure 22-9 LED Gauge Examples



- Rating: Displays and optionally accepts input for a metric value. This gauge is typically used to show ratings for products or services, such as the star rating for a movie.

Figure 22-10 shows five rating gauges configured with star, diamond, circle, rectangle and triangle shapes.

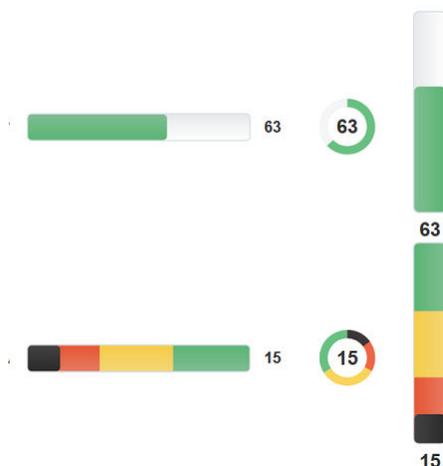
Figure 22-10 Rating Gauge Examples



- Status meter: Displays the metric value on a horizontal, vertical or circular axis. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Optionally, status meters can display colors to indicate where the metric value falls within predefined thresholds.

Figure 22-11 shows examples of status meter gauges configured as horizontal, circular and vertical status meters. The gauges are configured to use thresholds that display color to indicate whether the gauge's value falls within an acceptable range.

Figure 22-11 Status Meter Gauge Examples



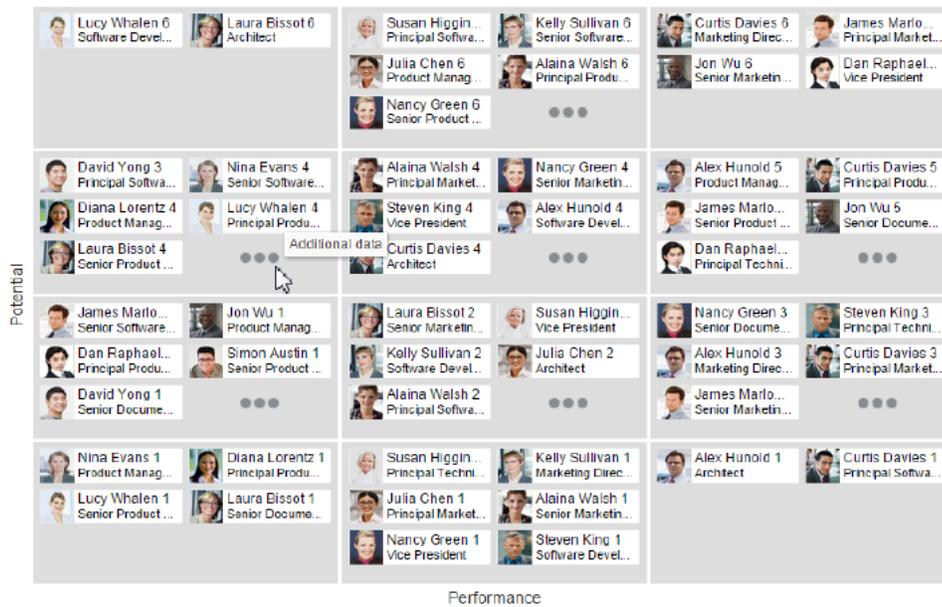
For more information including additional use cases and examples, see [Using Gauge Components](#).

NBox Use Cases and Examples

The `nBox` component is comprised of two parts: the node that represents the data and the grid that comprises the cells into which the nodes are placed. If the number of nodes is greater than the space allocated for the cell, the NBox displays an indicator that users can click to access the additional nodes.

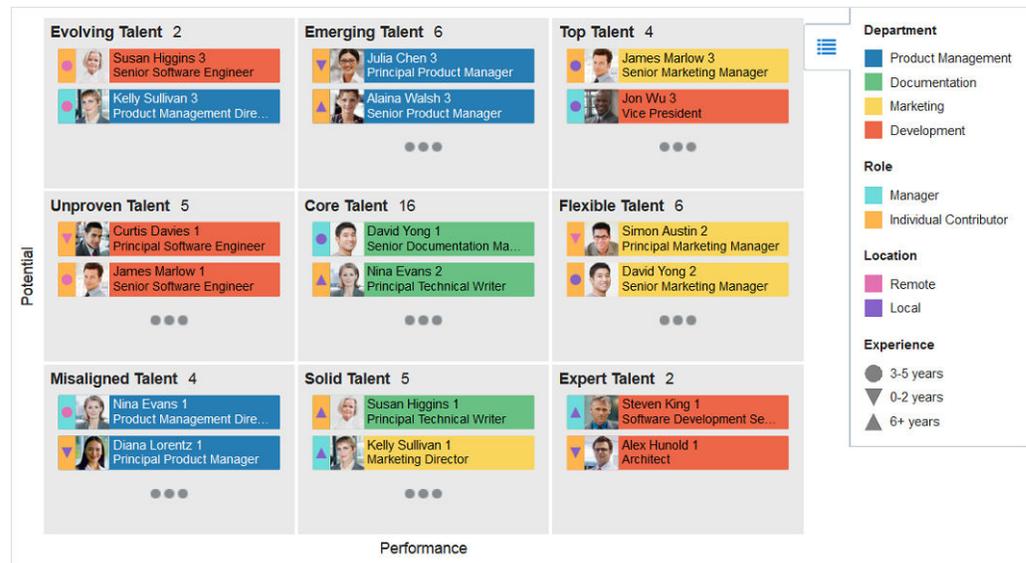
For example, as illustrated in [Figure 26-1](#) you can use the `nBox` component to compare employee potential and performance data, where the row represents employee potential and the column represents employee performance. The node that represents the employee is stamped into the appropriate cell.

Figure 22-12 NBox Component Comparing Employee Potential and Performance



NBox nodes can also be styled with colors, markers, and indicators to represent each unique value, or group, in the data set using attribute groups. [Figure 26-2](#) shows an NBox with employee nodes styled by department, role, and experience.

Figure 22-13 NBox Nodes Styled with Attribute Groups



NBox nodes representing attribute groups can also be configured to display by size and number within each grid cell as illustrated in [Figure 26-3](#), or across all cells are illustrated in [Figure 26-4](#).

Figure 22-14 NBox Nodes Displayed by Size and Number Within Cells

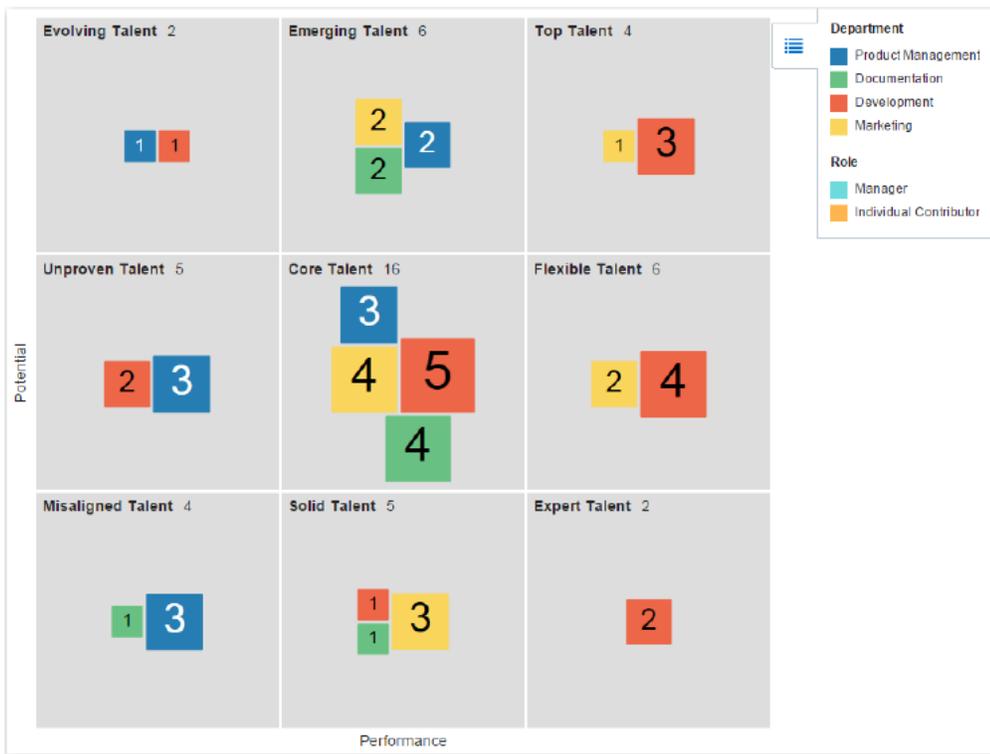


Figure 22-15 NBox Nodes Displayed by Size and Number Across Cells



Pivot Table Component Use Cases and Examples

The pivot table (`pivotTable`) produces a grid that supports multiple layers of data labels on rows or columns. An optional pivot filter bar (`pivotFilterBar`) can be associated with the pivot table to filter data not displayed in the row or column edge. When bound to an appropriate data control such as a row set, the component also supports the option of generating subtotals and totals for grid data, and drill operations at runtime.

Pivot tables let you swap data labels from one edge (row or column) or pivot filter bar (page edge) to another edge to obtain different views of your data. For example, a pivot table might initially display total sales data for products within regions on the row edge, broken out by years on the column edge. If you swap region and year at runtime, then you end up with total sales data for products within years, broken out by region.

Pivot tables support horizontal and vertical scrolling, header and cell formatting, and drag-and-drop pivoting. Pivot tables also support ascending and descending group sorting of rows at runtime. [Figure 22-16](#) shows an example pivot table with a pivot filter bar.

Figure 22-16 Pivot Table with Pivot Filter Bar

		World		Boston		Total Geography	
		Sales	Units	Sales	Units	Sales	Units
▼ 2005		8,750	35	500	9	9,250	44
	Canoes	3,750	20	375	2	4,125	22
	Tents	5,000	50	125	15	5,125	65
▼ 2006		17,500	70	1,000	15	18,500	85
	Canoes	7,500	40	750	4	8,250	44
	Tents	10,000	100	250	25	10,250	125
▼ 2007		15,000	28	900	11	15,900	39
	Tents	5,000	5	400	20	5,400	25
	Canoes	10,000	50	500	2	10,500	52
Average Year		6,875	44	400	11	7,275	56

For more information including additional use cases and examples, see [Using Pivot Table Components](#).

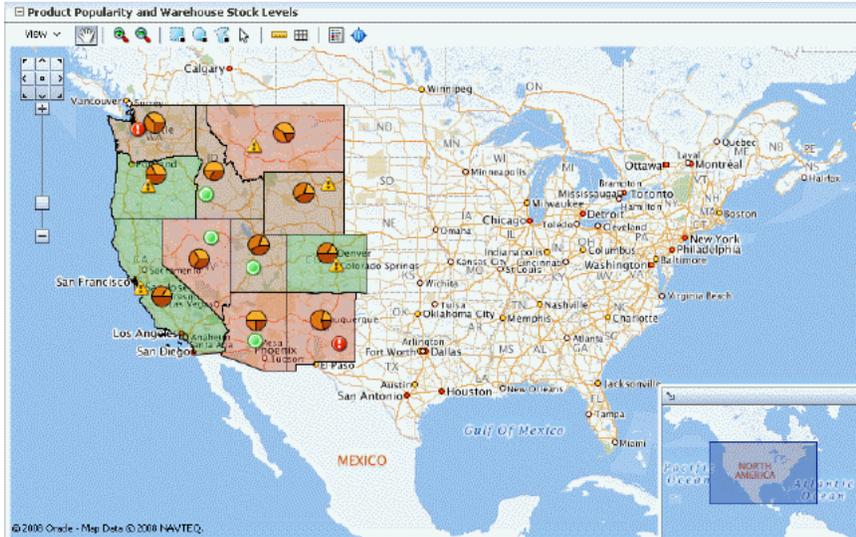
Geographic Map Component Use Cases and Examples

The geographic map (`map`) provides the functionality of Oracle Spatial within Oracle ADF. This component represents business data on a map and lets you superimpose multiple layers of information on a single map. This component supports the simultaneous display of a color theme, a graph theme (bar or pie graph), and point themes. You can create any number of each type of theme and you can use the map toolbar to select the desired themes at runtime.

As an example of a geographic map, consider a base map of the United States with a color theme that provides varying color intensity to indicate the popularity of a product within each state, a pie chart theme that shows the stock levels of warehouses, and a point theme that identifies the exact location of each warehouse. When all three themes are superimposed on the United States map, you can easily evaluate whether there is sufficient inventory to support the popularity level of a product in specific

locations. [Figure 22-17](#) shows a geographic map with color theme, pie graph theme, and point theme.

Figure 22-17 Geographic Map with Color Theme, Pie Graph Theme, and Point Theme



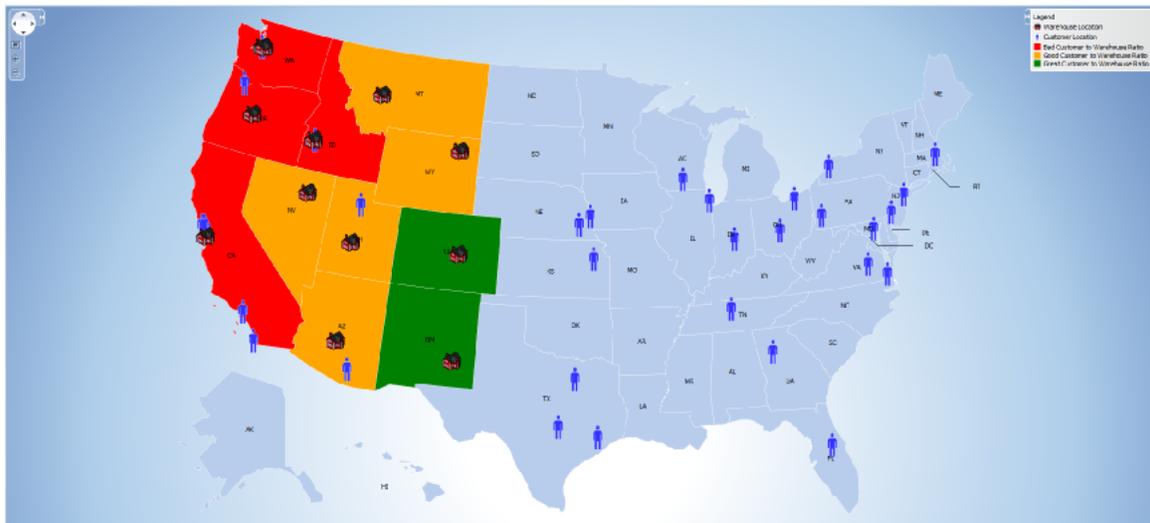
For more information including additional use cases and examples, see [Using Map Components](#).

Thematic Map Component Use Cases and Examples

A thematic map (`thematicMap`) component represents business data as patterns in stylized areas or associated markers and does not require a connection to a map viewer service. Thematic maps focus on data without the geographic details in a geographic map. The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, as well as base maps for continents and regions of the world such as EMEA and APAC. The thematic map component does not require a map service to display a base map.

For example, you could use a USA base map with a states map layer to display the location of warehouses and customers, with high, medium, and low ratios using colors as displayed in [Figure 22-18](#). The example illustrates thematic map default features including a data bound legend.

Figure 22-18 Thematic Map Displaying Warehouse to Customer Ratios



For more information including additional use cases and examples, see [Using Map Components](#) .

Gantt Chart Component Use Cases and Examples

The Gantt chart is a type of horizontal bar graph (with time on the horizontal axis) that is used in planning and tracking projects to show resources or tasks in a time frame with a distinct beginning and end.

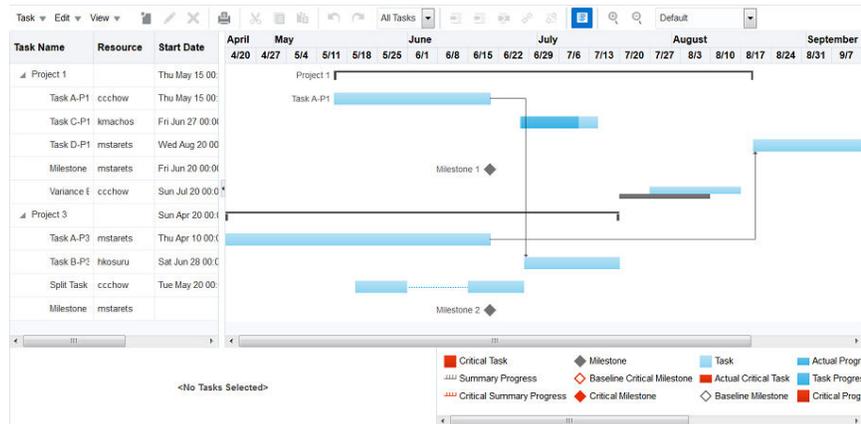
A Gantt chart consists of two ADF Faces tree tables combined with a splitter. The left-hand table contains a list of tasks or resources while the right-hand table consists of a single column in which progress is graphed over time.

There are three types of gantt components:

- Project Gantt (`projectGantt`): Creates a Gantt chart that shows tasks vertically, and the duration of the task is represented as a bar on a horizontal timeline.
- Resource utilization Gantt (`resourceUtilizationGantt`): Creates a Gantt chart that shows graphically whether resources are over or under allocated. It shows resources vertically while showing their allocation and, optionally, capacity on the horizontal time axis.
- Scheduling Gantt (`schedulingGantt`): Creates a Gantt chart that shows resource management and is based on manual scheduling boards. It shows resources vertically with corresponding activities on the horizontal time axis.

[Figure 22-19](#) shows a project Gantt view of staff resources and schedules.

Figure 22-19 Project Gantt Chart



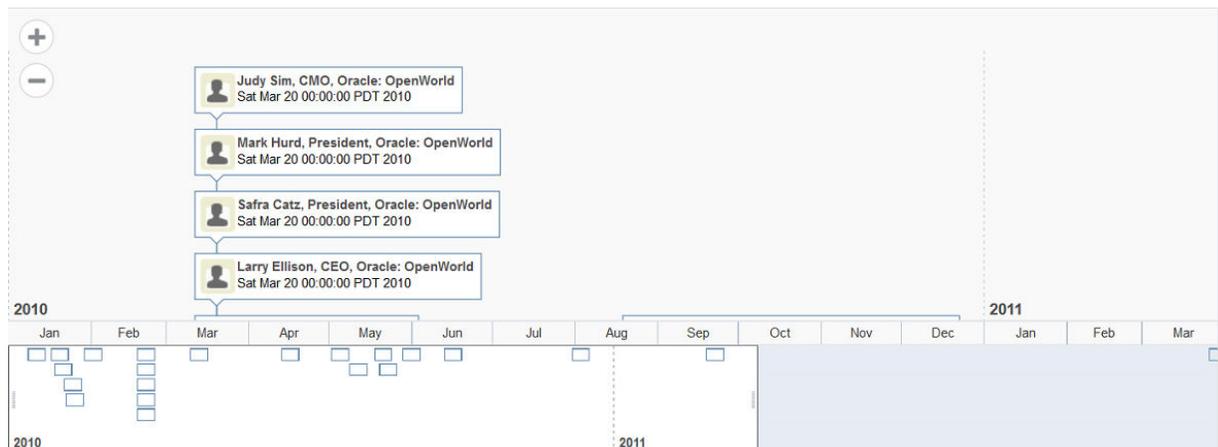
For more information including additional use cases and examples, see [Using Gantt Chart Components](#).

Timeline Component Use Cases and Examples

A timeline is composed of the display of events as timeline items along a time axis, a movable overview window that corresponds to the period of viewable time in the timeline, and an overview time axis that displays the total time increment for the timeline. A horizontal zoom control is available to change the viewable time range. Timeline items corresponding to events display related information or actions and are represented by a line feeler to the time axis and a marker in the overview time axis.

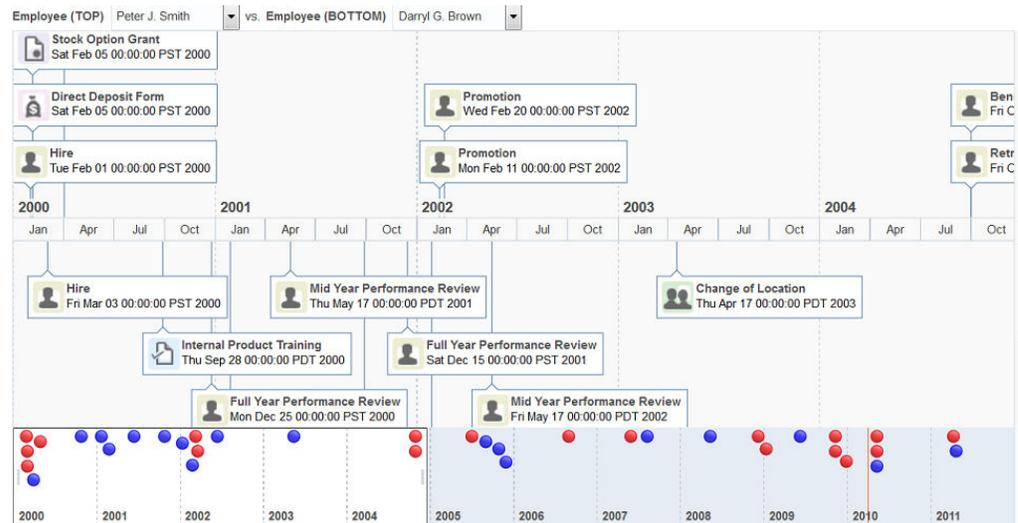
For example, the timeline in [Figure 22-20](#) is configured to display the chronological order of the hire dates of employees. In this example, timeline items representing each event display information about the employee using an image and text with labels. The overview window defines the time range for the display of the timeline items, adjustable by changing the zoom control or by changing the edges of the window to a larger or smaller size. When selection is configured, the timeline item, line feeler, and the event marker in the overview panel are highlighted.

Figure 22-20 Timeline of Employee Hire Dates



A dual timeline can be used for comparison of up to two series of events. [Figure 22-21](#) illustrates a dual timeline comparing employee change events for two employees over a ten year time period. Each event is represented in the respective employee's timeline as a card with a heading and the date and time as body text. Timeline events are displayed using a quarterly year time axis within the three plus year overview window. The red colored line in the overview time axis indicates the current date.

Figure 22-21 Dual Timeline Comparing Employee Change Events

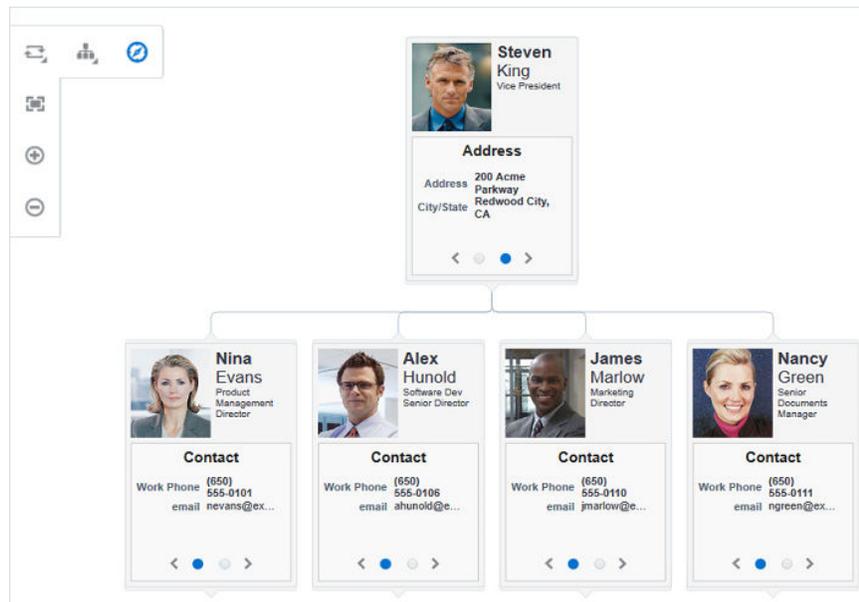


For more information including additional use cases and examples, see [Using Timeline Components](#) .

Hierarchy Viewer Component Use Cases and Examples

The hierarchy viewer (`hierarchyViewer`) component displays hierarchical data as a set of linked nodes in a diagram. The nodes and links correspond to the elements and relationships to the data. The component supports pan and zoom operations, expanding and collapsing of the nodes, rendering of simple ADF Faces components within the nodes, and search of the hierarchy viewer data. A common use of the hierarchy viewer is to display an organization chart with information about members in their respective nodes, as shown in [Figure 22-22](#).

Figure 22-22 Hierarchy Viewer as Organizational Chart



For more information including additional use cases and examples, see [Using Hierarchy Viewer Components](#).

Treemap and Sunburst Components Use Cases and Examples

The `treemap` and `sunburst` components display quantitative hierarchical data across two dimensions, represented visually by size and color. Treemaps and sunbursts use a shape called a `node` to reference the data in the hierarchy. For example, you can use a treemap or sunburst to display quarterly regional sales and to identify sales trends, using the size of the node to indicate each region's sales volume and the node's color to indicate whether that region's sales increased or decreased over the quarter.

Treemaps display nodes as a set of nested rectangles. Each branch of the tree is given a rectangle, which is then tiled with smaller rectangles representing sub-branches.

[Figure 22-23](#) shows a treemap displaying United States census data grouped by regions, with the color attribute used to indicate median income levels. States with larger populations display in larger-sized nodes than states with smaller populations.

hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information

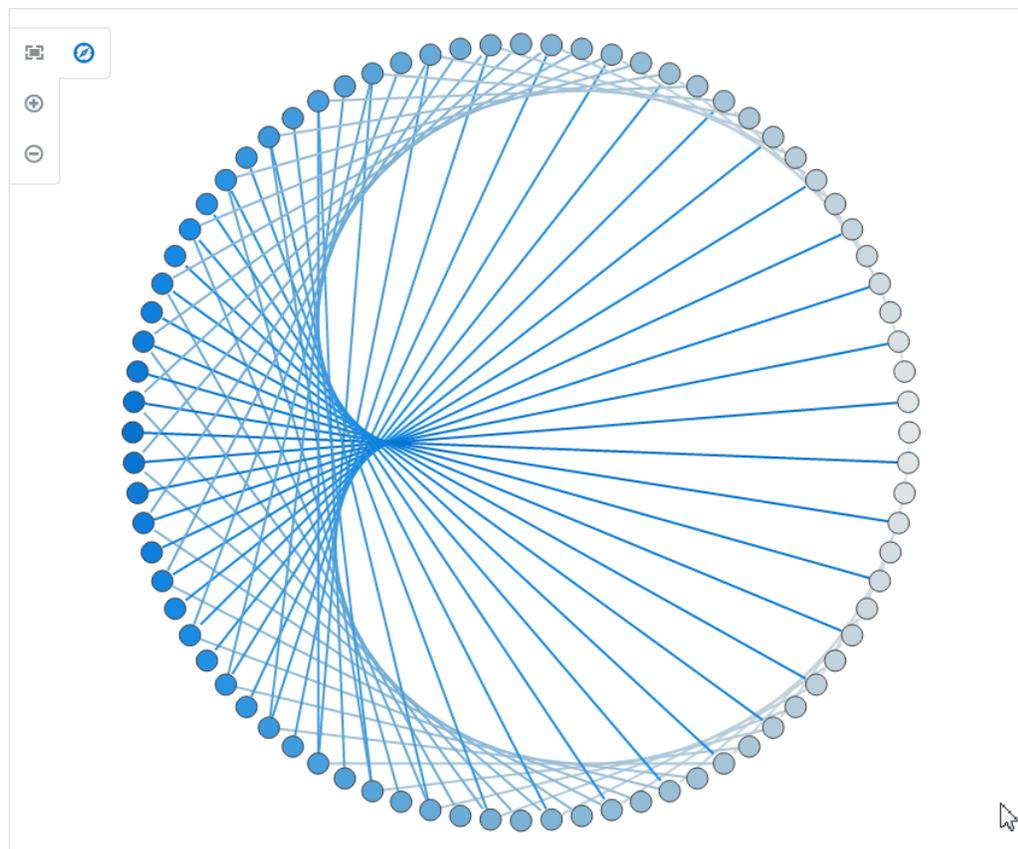
Use treemaps if you are primarily interested in displaying two metrics of data using size and color at a single layer of the hierarchy. Use sunbursts instead if you want to display the metrics for all levels in the hierarchy. Drilling can be enabled to allow the end user to traverse the hierarchy and focus in on key parts of the data.

For additional information about treemaps and how to use them in your application, see [Using Treemap and Sunburst Components](#).

Diagram Use Cases and Examples

Diagrams use shapes for the node, and lines for links to represent the relationships between the nodes. [Figure 33-1](#) shows a simple diagram configured in a circular layout pattern with circles and lines representing the relationships between them. The example includes the default control panel for diagram zooming.

Figure 22-25 Simple Diagram with Nodes and Links with Control Panel



Diagrams can also be configured to visually display hierarchical data with a master-detail relationship. [Figure 33-2](#) shows an employee tree diagram at runtime that includes a control panel, a number of nodes, and links that connect the nodes. Also illustrated is a node panel card that uses `af:showDetailItem` elements to display multiple sets of data at different zoom levels.

Tag Cloud Component Use Cases and Examples

Tag Clouds are made up of an array of tags. There are a number of use cases.

Tag clouds are most useful to quickly gauge the relative percentage power of an item in a population. For example, [Figure 22-30](#) shows the results of a survey wherein respondents disclosed which social media networks they use.

Figure 22-30 Simple Tag Cloud with Cloud Layout



Tag Clouds may also be displayed in a rectangular layout. This is most useful for formal data, website tag searches, or if the cloud needs to be contained within a fixed container. [Figure 22-31](#) shows the latest census data for population of the states in the US. The tags are configured to show the population in the tooltip for the tag.

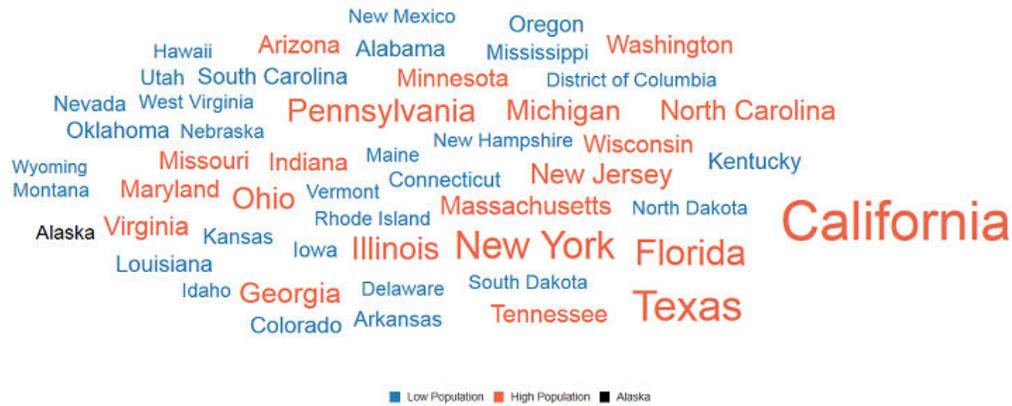
Figure 22-31 Tag Cloud with Rectangular Layout



Tags may be divided into groups, and these groups may be represented in a legend. This is useful for quickly dividing tags by criteria. For example, [Figure 22-32](#) shows US

states bifurcated into groups of low or high population. The condition provided classifies states with less than 5 million people as low population and with more than 5 million people as high population. Tag clouds accept both match and exception rules to allow for complex grouping.

Figure 22-32 Tag Cloud with Colored Groups and Legend



Additional Functionality for Data Visualization Components

You may find it helpful to understand other ADF Faces features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a data visualization component to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: Users can change the way the data visualization components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: Data visualization components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Skins and styles: You can customize the appearance of data visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Placeholder data controls: If you know the data visualization components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder

data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Common Functionality in Data Visualization Components

ADF Data Visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), the image format used to display the component, and how data can be displayed and edited.

It is important that you understand this shared functionality and how it is configured before you use these components.

Content Delivery

Data visualization components including chart, gauge, Gantt chart, hierarchy viewer, pivot table, sunburst, thematic map, timeline, and treemap can be configured for how data is delivered from the data source. The data can be delivered to the components either immediately upon rendering, as soon as the data is available, or lazily fetched after the shell of the component has been rendered. By default all data visualization components, with the exception of the geographic map, support the delivery of content from the data source when it is available. The `contentDelivery` attribute of these components is set to `whenAvailable` by default.

Data visualization components based on a tree or tree table model including Gantt charts, hierarchy viewers, pivot tables, sunbursts, timelines, and treemaps are virtualized, meaning not all the rows, columns, or levels that are there for the component on the server are delivered to and displayed on the client. You configure these components to fetch a certain number of rows, columns, or levels at a time from your data source. Use these attributes to configure fetch size:

- Gantt charts:
 - `fetchSize`: Specifies the number of rows in the data fetch block. The default value is 25.
 - `horizontalFetchSize`: Specifies the size of the horizontal data window in number of pixels in which the data are fetched. Only task bars within this data window would be rendered. In contrast with `fetchSize`, which provides vertical virtualization, `horizontalFetchSize` provides horizontal virtualization.
- Hierarchy Viewer:
 - `levelFetchSize`: Specifies the number of child nodes that will be fetched and displayed at a single time for each expanded parent node. Additional child nodes may be fetched and displayed by using the lateral navigation controls shown in the hierarchy viewer. The default value is 25.
- Pivot table:
 - `rowFetchSize`: Specifies the number of rows in a data fetch block. The default value is 25.
 - `columnFetchSize`: Specifies the number of columns in a data fetch block. The default value is 10.

- Sunburst:
 - `displayLevelsChildren`: Specifies the number of child levels to display during initial render. This property is 0-based. A value of 0 means that no child levels below the root will be shown; the root itself will be shown. The default value is 2, which means that the root and the first two levels of children will be shown.
- Timeline:
 - `fetchStartTime`: Specifies the start of the time range where data is currently being fetched
 - `fetchEndTime`: Specifies the end of the time range where data is currently being fetched.
- Treemap:
 - `displayLevelsChildren`: Specifies the number of child levels to display during initial render. This property is 0-based. A value of 0 means that no child levels below the root will be shown; the root itself will be shown. The default value is 2, which means that the root and the first two levels of children will be shown.

For lazy delivery, when a page contains one or more of these components, the page initially goes through the standard life cycle. However, instead of fetching the data during that initial request, a special separate partial page rendering (PPR) request is run, and the value of the fetch size for the component is then returned. Because the page has just been rendered, only the Render Response phase executes for the components, allowing the corresponding data to be fetched and displayed. When a user's actions cause a subsequent data fetch (for example scrolling in a pivot table grid for another set of rows), another PPR request is executed.

When content delivery is configured to be delivered when it is available, the framework checks for data availability during the initial request, and if it is available, it sends the data to the component. If it is not available, the data is loaded during the separate PPR request, as it is with lazy delivery.

 **Performance Tip:**

Lazy delivery should be used when a data fetch is expected to be an expensive (slow) operation, for example, slow, high-latency database connection, or fetching data from slow data sources like web services. Lazy delivery should also be used when the page contains a number of components other than a data visualization component. Doing so allows the initial page layout and other components to be rendered first before the data is available.

Immediate delivery should be used if the data visualization component is the only context on the page, or if the component is not expected to return a large set of data. In this case, response time will be faster than using lazy delivery (or in some cases, simply perceived as faster), as the second request will not go to the server, providing a faster user response time and better server CPU utilizations. Note that for components based on a tree or tree table model, only the value configured to be the fetch block will be initially returned. As with lazy delivery, when a user's actions cause a subsequent data fetch, the next set of rows are delivered.

The `whenAvailable` delivery provides the additional flexibility of using immediate when data is available during initial rendering or falling back on lazy when data is not initially available.

For more information about setting the fetch size for components based on the tree or tree table model, see [Content Delivery](#).

Automatic Partial Page Rendering (PPR)

ADF Faces supports Partial Page Rendering (PPR), which allows certain components on a page to be rerendered without the need to rerender the entire page. In addition to built-in PPR functionality, you can configure components to use cross-component rendering, which allows you to set up dependencies so that one component acts as a trigger and another as the listener. For more information, see [About Partial Page Rendering](#).

By default, ADF Data Visualization components support automatic PPR, where any component whose values change as a result of backend business logic is automatically rerendered. If your application uses the Fusion technology stack, you can enable the automatic partial page rendering feature on any page. For more information, see the "What You May Need to Know About Partial Page Rendering and Iterator Bindings" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Active Data Support

The Fusion technology stack includes the Active Data Service (ADS), which is a server-side push framework that allows you to provide real-time data updates for ADF Faces components and ADF Data Visualization components. You bind ADF Faces components to a data source and ADS pushes the data updates to the browser client without requiring the browser client to explicitly request it.

Table 22-1 lists the DVT components that support active data and where you can find additional detail.

Table 22-1 DVT Components Supporting Active Data

DVT Component	Link to Component Detail
geographic map	What You May Need to Know About Active Data Support for Map Point Themes
pivot table and pivot filter bar	Active Data Support (ADS) Supports ADS only when the <code>outputText</code> component or <code>sparkChart</code> is configured to display the active data; other components are not supported inside collection-based component.
sunburst	Active Data Support (ADS)
treemap	Active Data Support (ADS)
chart (all types)	Active Data Support (ADS)
gauge (all types)	Active Data Support (ADS)

For additional information about using the Active Data Service, see [Using the Active Data Service with an Asynchronous Backend](#).

Text Resources from Application Resource Bundles

JDeveloper supports easy localization of ADF Faces and data visualization components using the abstract class `java.util.ResourceBundle` to provide locale-specific resources.

Data visualization components may include text that is part of the component, for example the `af:table` component uses the resource string `af_table.LABEL_FETCHING` for the message text that displays in the browser while the `af:table` component fetches data during the initial load of data or while the user scrolls the table. JDeveloper provides automatic translation of these text resources into 28 languages. These text resources are referenced in a resource bundle. If you set the browser to use the language in Italy, any text contained within the components will automatically be displayed in Italian.

For any text you add to a component, for example if you define the title of a `pieChart` component by setting its `title` attribute, you must provide a resource bundle that holds the actual text, create a version of the resource bundle for each locale, and add a `<locale-config>` element to define default and support locales in the application's `faces-config.xml` file. You must also add a `<resource-bundle>` element to your application's `faces-config.xml` file in order to make the resource bundles available to all the pages in your application. Once you have configured and registered a resource bundle, the Expression Language (EL) editor will display the key from the bundle, making it easier to reference the bundle in application pages.

To simplify the process of creating text resources for text you add to ADF components, JDeveloper supports automatic resource bundle synchronization for any translatable string in the visual editor. When you edit components directly in the visual editor or in the Properties window, text resources are automatically created in the base resource bundle. For more information, see [Using Automatic Resource Bundle Integration in JDeveloper](#).

 **Note:**

Any text retrieved from the database is not translated.

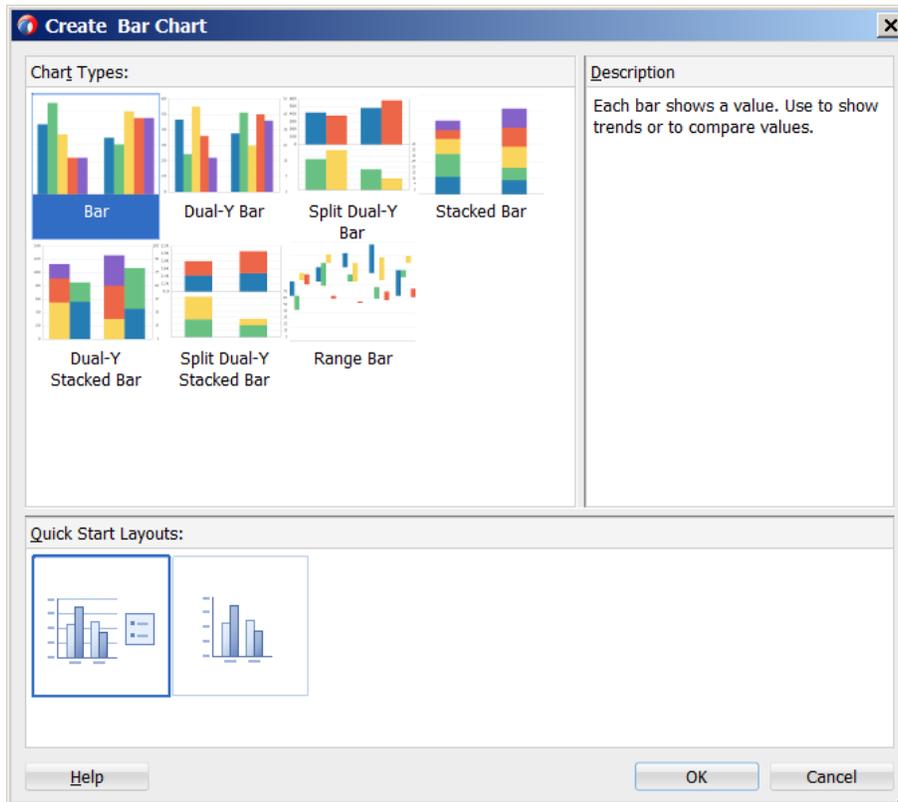
For data visualization components with title and label child components, you can also create and add text resources to a resource bundle by using the attribute dropdown list to open a Select Text Resource dialog to select or add a translatable string from an application resource bundle. Alternatively, you can select Expression Builder to open the Expression Language (EL) editor to create an expression to be executed at runtime for the title or label.

Providing Data for ADF Data Visualization Components

In JDeveloper you can add any ADF Data Visualization component to your JSF page using UI-first development, and then later manually bind the data you wish to display using ADF data controls or managed beans. In this case you drag the component from the Components window to the page and manually bind the data in the Properties window.

For example, when you are designing your page using simple UI-first development, you use the Components window to add a bar chart to a JSF page. When you drag and drop a chart component onto the page, a Create Chart dialog displays available categories of chart types, with descriptions, to provide visual assistance when creating charts. You can also specify a quick start layout of the chart's legend. [Figure 22-33](#) shows the Create Bar Chart dialog for bar charts with the default bar chart type and quick start layout selected.

Figure 22-33 Create Bar Chart Dialog



For information about creating Data Visualization components using UI-first development, understanding component data requirements, configuring DVT parent and child components, customizing the appearance of components, and adding special effects and interactivity to components, see the following chapters in this part of the guide:

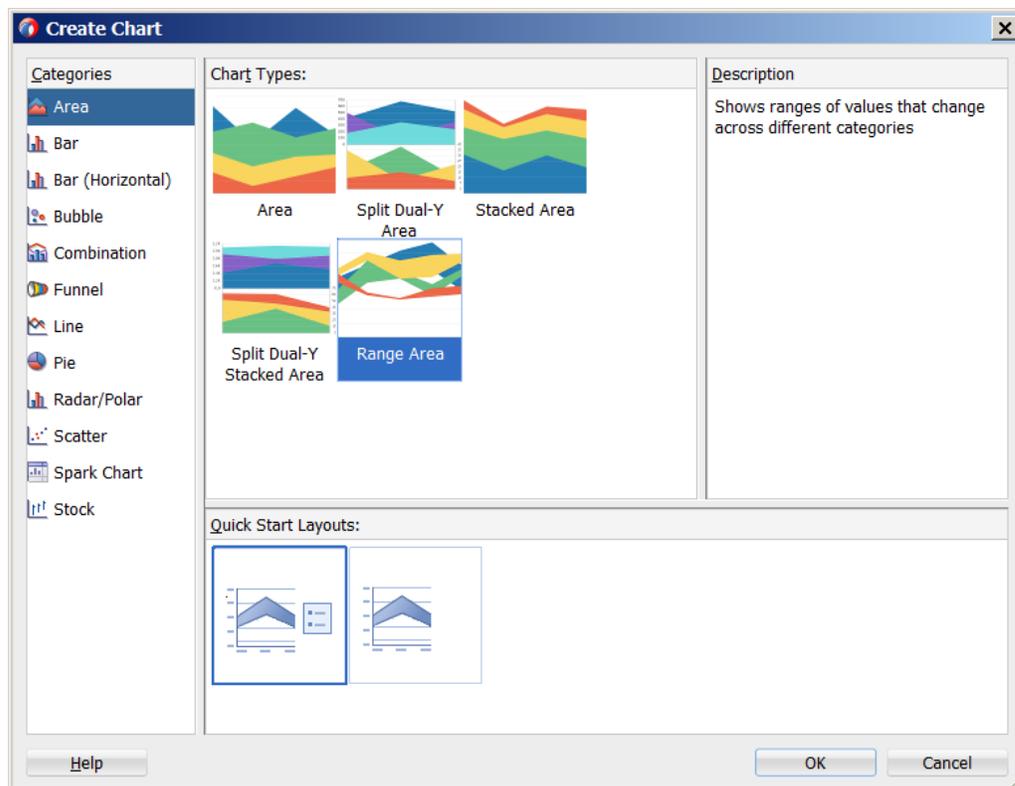
- [Using Chart Components](#)
- [Using Picto Chart Components](#)
- [Using Gauge Components](#)
- [Using NBox Components](#)
- [Using Pivot Table Components](#)
- [Using Gantt Chart Components](#)
- [Using Timeline Components](#)
- [Using Map Components](#)
- [Using Hierarchy Viewer Components](#)
- [Using Treemap and Sunburst Components](#)
- [Using Diagram Components](#)
- [Using Tag Cloud Components](#)

Alternatively, you can use data-first development and create the component using an ADF data control that will handle the data binding for you. In this case you drag a data

collection from the Data Controls panel and complete the data binding dialogs to configure the display of data.

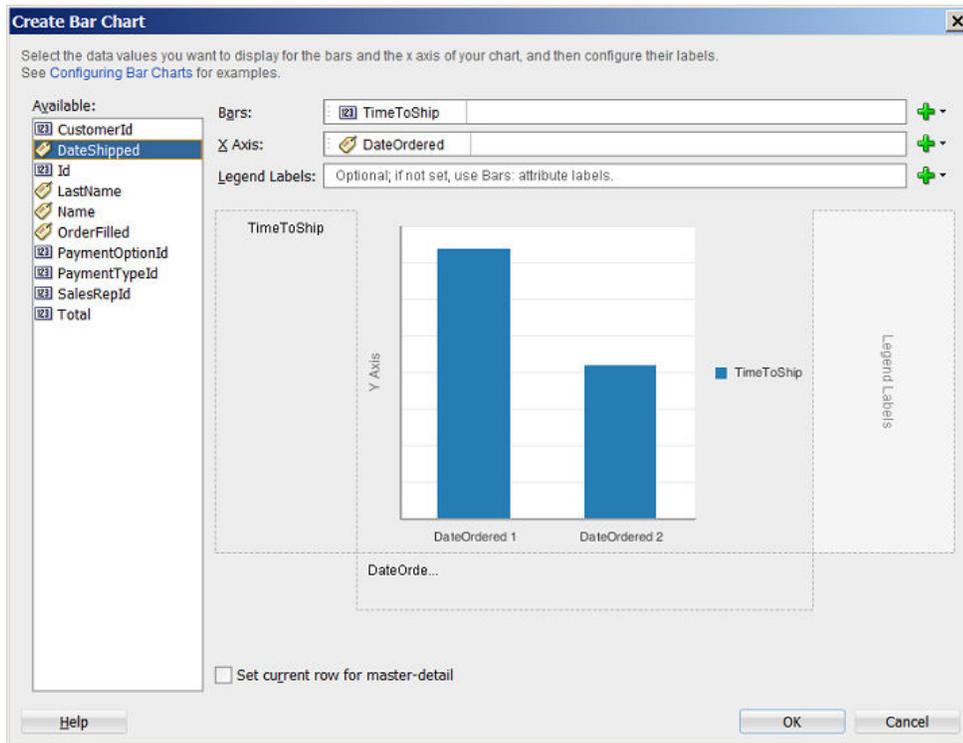
For example, you can create and data bind a DVT chart by dragging a data control from the Data Controls Panel. A Component Gallery displays available chart categories, types, and descriptions to provide visual assistance when designing charts and defining a quick layout. [Figure 22-34](#) shows the Component Gallery that displays when creating a chart from a data control.

Figure 22-34 Component Gallery for Charts



After selecting the category and type of chart you wish to create, a data binding dialog is displayed to bind the data collection attributes to the chart component. [Figure 22-35](#) shows the Create Bar Chart dialog used to create and data bind a bar chart.

Figure 22-35 Data Binding in the Create Bar Chart Dialog



All data visualization components can be bound to data collections in an ADF data control. For information and examples of data binding these components to data controls, see the following:

- Creating Databound Charts section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Picto Charts section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Gauges section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound NBox Components section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Pivot Tables section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

In JDeveloper, a **Create Pivot Table** wizard provides declarative support for data-binding and configuring the pivot table.

- Creating Databound Geographic Maps section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Creating Databound Thematic Maps section in *Developing Fusion Web Applications with Oracle Application Development Framework*
- Creating Databound Gantt Charts section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Timelines section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Hierarchy Viewer section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Treemaps and Sunbursts section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Diagram Components section in *Developing Fusion Web Applications with Oracle Application Development Framework*.
- Creating Databound Tag Clouds section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

23

Using Chart Components

This chapter describes how to use the ADF Data Visualization chart components to display data in charts using simple UI-first development. The chart components include area, bar, bubble, combination, funnel, line, pie, scatter, spark, and stock charts. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of these components.

If your application uses the Fusion technology stack, you can use data controls to create charts. For more information, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Chart Component](#)
- [Using the Chart Component](#)
- [Adding Data to Charts](#)
- [Customizing Chart Display Elements](#)
- [Adding Interactive Features to Charts](#)

About the Chart Component

The ADF DVT chart components are a set of visualizations for evaluating data points on multiple axes. They present a variety of forms that are useful in many ways, such as comparing results from two or more groups, showing stock information and presenting a pie chart comparison.

Charts display series and groups of data. Series and groups are analogous to the rows and columns of a grid of data. Typically, the rows in the grid appear as a series in a chart, and the columns in the grid appear as groups.

For most charts, a series appears as a set of markers that are the same color. Typically, the chart legend shows the identification and associated color of each series. For example, in a bar chart, the yellow bars might represent the sales of shoes and the green bars might represent the sales of boots.

Groups appear differently in different chart types. For example, in a stacked bar chart, each stack is a group. A group might represent time periods, such as years. A group might also represent geographical locations such as regions.

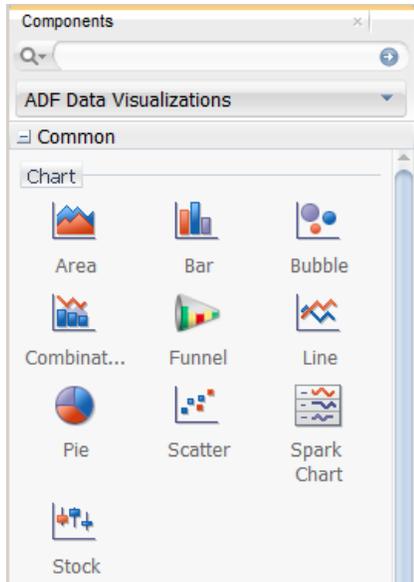
Depending on the data requirements for a chart type, a single data item might require one or more data values. For example, a scatter chart requires two values for each data marker. The first value determines where the marker appears along the x-axis while the second value determines where the marker appears along the y-axis.

Chart Component Use Cases and Examples

The chart components include ten types of charts with one or more variations that you can use to display data. JDeveloper provides a Components window that displays

available chart categories. [Figure 23-1](#) shows the Components window for area, bar, bubble, combination, funnel, line, pie, scatter, spark and stock charts.

Figure 23-1 Components Window for Charts



When you select a chart category in the Components window, JDeveloper displays a dialog with descriptions about the available chart types to provide visual assistance when you are creating charts. [Figure 23-2](#) shows the different area chart types and layouts available when you select the `Area` chart in the Components window.

Figure 23-2 Area Chart Types in Create Area Chart Dialog

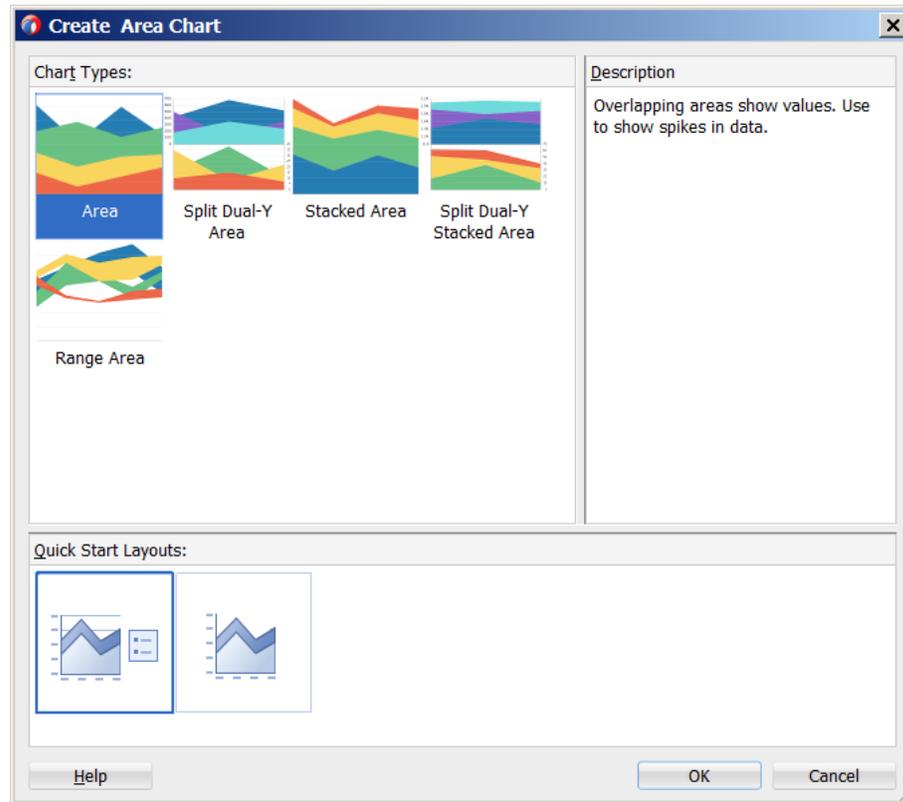


Chart categories include:

- **Area:** Represents data as a filled-in area. Use area charts to show trends over time, such as sales for the last 12 months. Area charts require at least two groups of data along an axis. The axis is often labeled with increments of time such as months.

Area charts represent these kinds of data values:

- **Absolute:** Each area marker connects a series of two or more data values.
- **Stacked:** Area markers are stacked. The values of each set of data are added to the values for previous sets. The size of the stack represents a cumulative total.

 **Tip:**

Stacked charts are generally preferred over absolute charts. Areas in absolute charts can be visually obscured by other areas, depending on the area's data value.

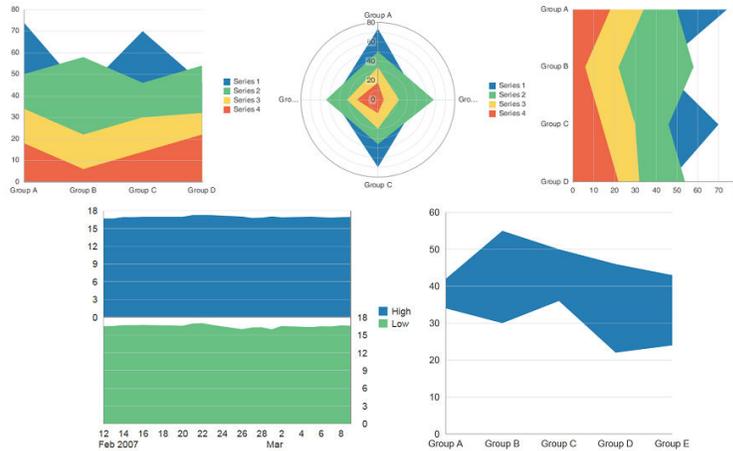
- **Split Dual-Y:** Dual axis charts are split into two parts. Area markers unique to a particular axis are plotted only on the relevant part. Applies to both absolute and stacked charts.

- Range: Area markers and area coverage are determined by high and low values.

All variations of area charts can be configured with a single y-axis or dual y-axis.

Figure 23-3 shows example area charts with Cartesian and polar coordinate systems, as well as a horizontal area chart. It also shows sample split Dual-Y area and range area charts.

Figure 23-3 Area Chart Example



- Bar: Represents data as a series of vertical bars. Use bar charts to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.

Bar charts represent these kinds of data values:

- Clustered: Each cluster of bars represents a group of data. For example, if data is grouped by employee, one cluster might consist of a Salary bar and a Commission bar for a given employee. This kind of chart includes the following variations: vertical clustered bar charts and horizontal clustered bar charts.
- Stacked: Bars for each set of data are appended to previous sets of data. The size of the stack represents a cumulative data total.
- Split Dual-Y: Dual-axis charts are split into two parts. Bars unique to a particular axis are plotted only on the relevant part. Applies to both absolute and stacked charts.
- Range: Bars do not originate from the X-axis. The beginning and end of a bar are determined by high and low values. Useful to create waterfall charts for financial tracking.

All variations of bar charts can be configured with a single y-axis or dual y-axis. Bar charts may also be displayed horizontally using an orientation attribute.

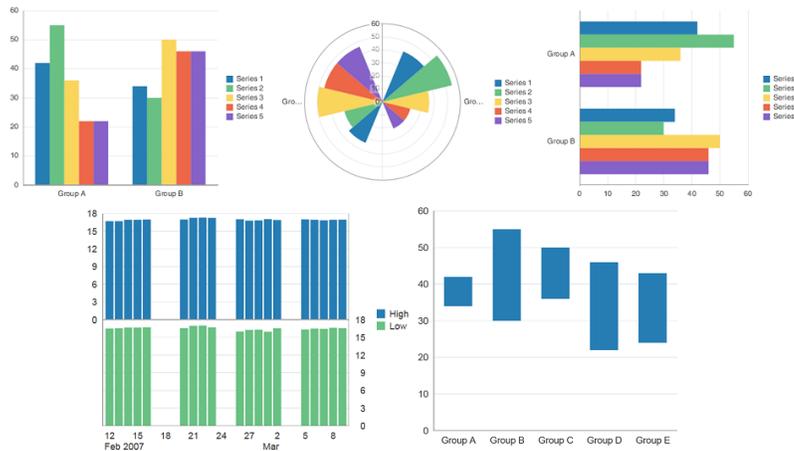
Figure 23-4 shows variations of the bar chart type as displayed in the Create Bar Chart dialog with the Dual-Y Stacked Bar chart selected.

Figure 23-4 Bar Chart Types



Figure 23-5 shows example bar charts with Cartesian and polar coordinate systems, as well as a horizontal bar chart. It also shows sample split Dual-Y bar and range bar charts.

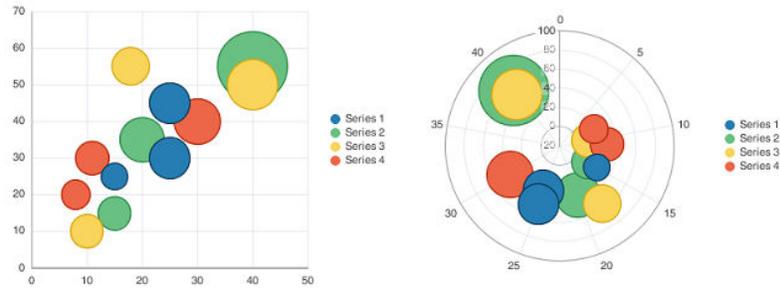
Figure 23-5 Bar Chart Example



- **Bubble:** Represents data by the location and size of round data markers (bubbles). Use bubble charts to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble chart to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a chart allows you to examine productivity relative to salary and experience.

Figure 23-6 shows example bubble chart with Cartesian and polar coordinate systems.

Figure 23-6 Bubble Chart Example



- **Combination:** Chart that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use combination charts to compare bars and lines, bars and areas, lines and areas, or all three combinations. Combination charts may also be stacked to represent cumulative totals.

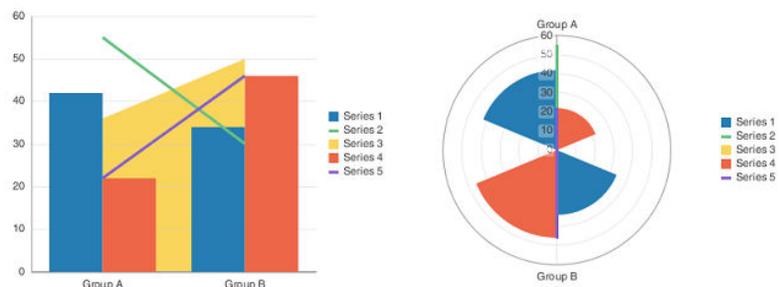
Figure 23-7 shows variations of the combination chart type as displayed in the Create Combination Chart dialog with the default combination chart selected. Combination charts can be configured with a single y-axis or dual y-axis, and can be stacked. Dual y-axis combination charts can be configured with a split axis.

Figure 23-7 Combination Chart Types



Figure 23-8 shows example combination charts with Cartesian and polar coordinate systems.

Figure 23-8 Combination Chart Example

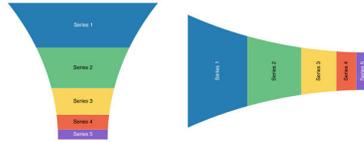


- **Funnel:** Visually represents data related to steps in a process. The steps appear as slices across a vertical or horizontal cone-shaped section. Typically, a funnel chart requires actual values against a stage value, which might be time. An optional target value may be specified for comparison. If target values are

specified, a given step or slice fills as its actual value approaches its quota. For example, use the funnel chart to watch a process where the different sections of the funnel represent different stages in the sales cycle.

Figure 23-9 shows example vertical and horizontal funnel charts.

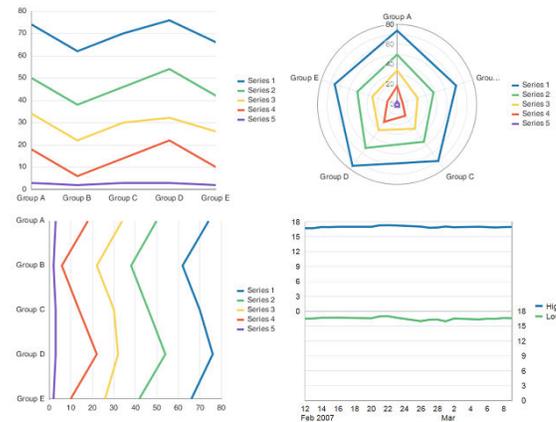
Figure 23-9 Funnel Chart Example



- **Line:** Represents data as a line, as a series of data points, or as data points that are connected by a line. Line charts require data for at least two points for each member in a group. For example, a line chart over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, and Asia. Use line charts to compare items over the same time. Line charts can be configured with a single y-axis or dual y-axis. The dual y-axis may be configured as a Split Dual-Y line chart.

Figure 23-10 shows example line charts with Cartesian and polar coordinate systems, as well as a horizontal line chart and a sample split dual-y line chart.

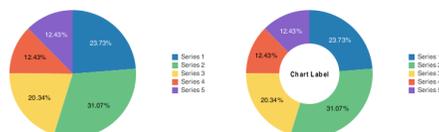
Figure 23-10 Line Chart Example



- **Pie:** Represents a set of data items as proportions of a total. The data items are displayed as sections of a circle causing the circle to look like a sliced pie. Use pie charts to show the relationship of parts to a whole such as how much revenue comes from each product line. A pie chart may also be customized as a donut chart, which has an empty area in the center of the chart.

Figure 23-11 shows an example pie chart, as well as a ring/donut chart, with a chart label.

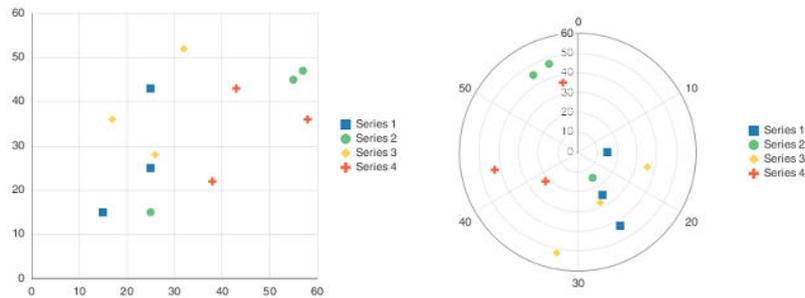
Figure 23-11 Pie Chart Example



- Scatter: Represents data by the location of data markers. Use scatter charts to show correlation between two different kinds of data values such as sales and costs for top products. Use scatter charts in particular to see general relationships among a number of items.

Figure 23-12 shows example scatter charts with Cartesian and polar coordinate systems.

Figure 23-12 Scatter Chart Example



- Spark: A simple, condensed chart that displays trends or variations in a single data value, typically stamped in the column of a table or in line with related text. Spark charts have basic conditional formatting. Since spark charts contain no labels, the adjacent columns of a table or surrounding text provide context for spark chart content

Figure 23-13 shows variations of the spark chart type as displayed in the Create Spark Chart dialog with the default chart selected.

Figure 23-13 Spark Chart Types

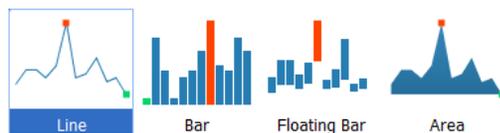


Figure 23-14 shows examples of line, bar, area, and floating bar spark charts.

Figure 23-14 Spark Chart Example



- Stock: Represents information related to trade stocks, specifically the opening, high, low, and closing prices of stocks. Use stock charts to explore trends in stock prices, fluctuations and volume of trade. Depending on the type of chart chosen, each stock marker displays two to four values, not counting the optional volume marker. When the volume of trading is included, the volume appears as bars in the lower part of the chart.

Figure 23-15 shows variations of the stock chart type as displayed in the Create Stock Chart dialog with the Open Close Candle chart selected.

Figure 23-15 Stock Chart Types

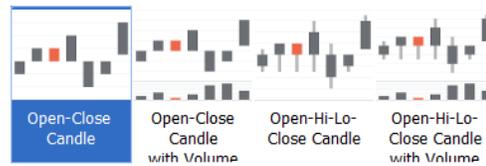


Figure 23-16 shows an example stock chart, with four data items.

Figure 23-16 Stock Chart Example



End User and Presentation Features of Charts

Chart end user and configurable presentation features include a rich variety of options.

Chart Data Labels

Use data labels to display information about the data points. You can customize the text, position, and style.

Figure 23-17 shows a bubble chart and a scatter chart, each configured to show data labels. In the bubble chart, the group's value is displayed in the center of the bubble. In the scatter chart, the label position varies by series. In the first series, the label is positioned below the series marker, and the second series is set to `auto` which displays the label after the marker. The third series label is configured to display the label before the series marker, and the fourth series label displays the label above the marker.

Figure 23-17 Chart Data Labels

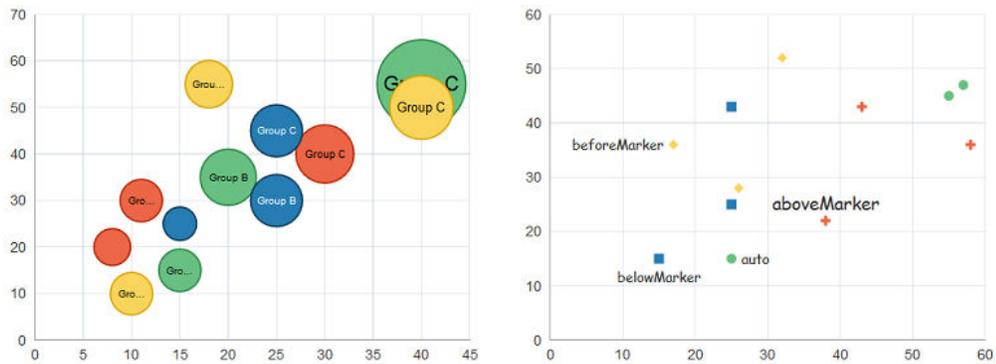


Chart Element Labels

You can add descriptive labels to most chart components and subcomponents, including titles, subtitles, axis labels, footnotes, and legends.

Figure 23-18 shows a bar chart configured to show a title and subtitle. The chart is also configured to show titles for the x-axis, y-axis, legend, and footnote.

Figure 23-18 Bar Chart Configured with Labels for Chart, X-Axis, Y-Axis, and Legend

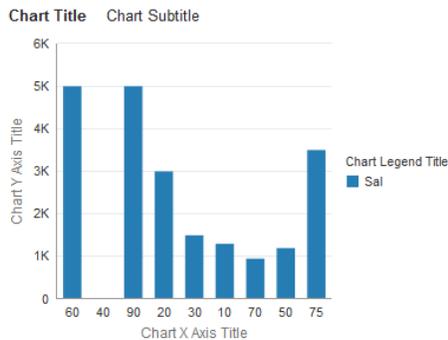
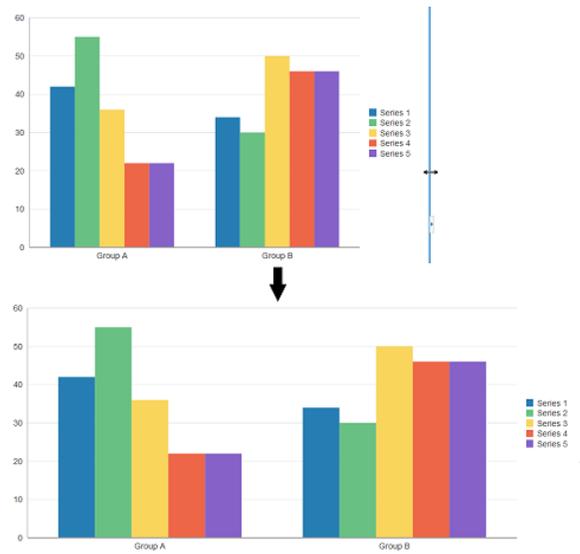


Chart Sizing

Charts use client-side layout management for controlling the size of the chart. A chart can automatically adjust to the size of the chart's container, and the user can resize the chart by resizing its container. You can also specify the size of a chart using its `inlineStyle` or `styleClass` attributes.

Figure 23-19 shows a portion of a page configured with the `af:panelSplitter` and `dvt:barChart` components. The user can drag the splitter to change the bar chart's size.

Figure 23-19 Bar Chart Resized by Dragging a Panel Splitter



When charts are displayed in a horizontally or vertically restricted area, as in a web page sidebar, the chart is displayed in a fully featured, although simplified display.

Chart Legends

Chart legends identify the chart's series and associated colors. [Figure 23-20](#) shows a bar chart configured with a legend. In this example, the number of series is greater than the legend area, and the user can scroll through the legend items to see all series on the chart.

Figure 23-20 Bar Chart With Scrollable Legend

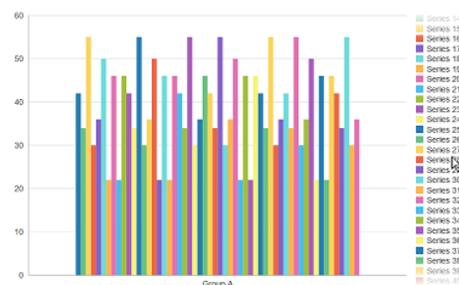


Chart Styling

Charts support styling of colors, sizes, and text to customize series, markers, lines, and data items. [Figure 23-21](#) shows an area chart configured with custom colors for the series items and borders and a scatter chart configured with custom markers.

Figure 23-21 Charts Showing Styling for Colors and Markers

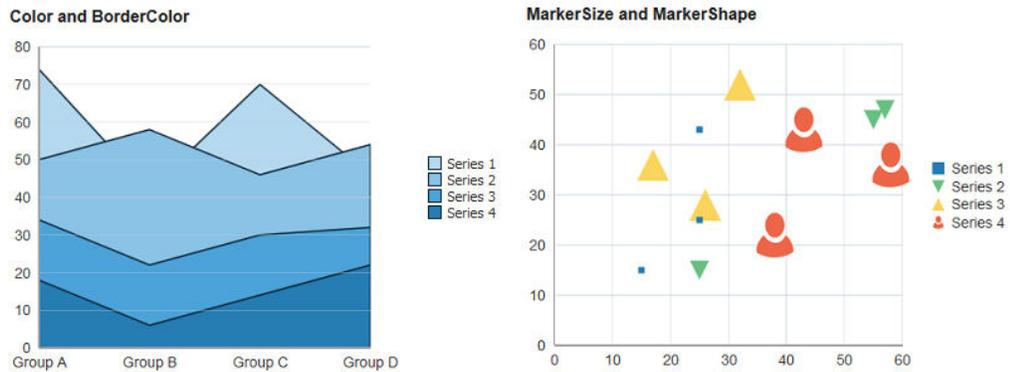


Chart Series Hiding

You can configure charts to allow the user to click on a legend series item to hide a series item from view. The resulting chart can be rescaled or rendered without rescaling.

Figure 23-22 shows a bar chart configured for series show and hide. When the user clicks a series item in the chart legend, the series no longer renders, and the legend changes to show which series item is hidden from view. The user can click the series item again to restore the series view.

Figure 23-22 Bar Chart Configured for Series Show and Hide

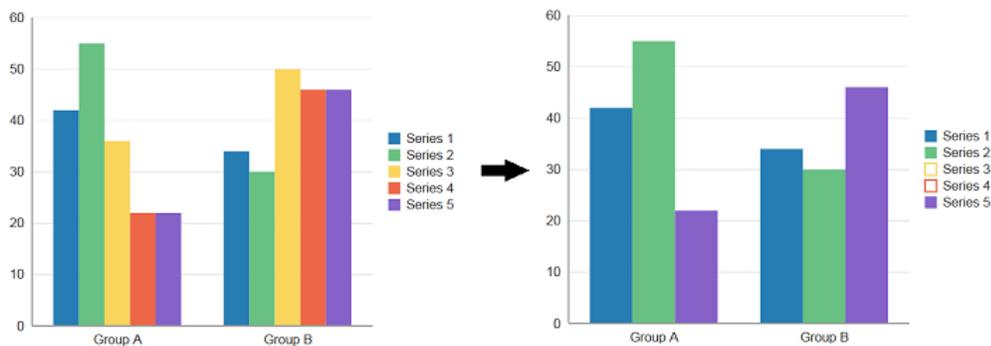


Chart Reference Objects

You can add reference lines or areas to a specified location or area on a chart's axis.

Figure 23-23 shows an example of a bar chart configured to show a reference line and reference area along its y-axis. In this example, the chart is configured to display a dark blue reference line at 25 on the y-axis. The reference area is configured to display in blue all values between a minimum of 95 and a maximum of 140.

Figure 23-23 Bar Chart Configured With Reference Line and Reference Area



Chart Series Effects

By default, charts apply gradients to chart series. You can remove the gradients to achieve a flatter design or display the series with patterns. Figure 23-24 shows three area charts configured for series effects.

Figure 23-24 Area Charts Configured for Series Effects



Chart Series Customization

You can customize the appearance of individual series in a chart. Depending upon the chart type, you can customize colors, markers, lines, and fill effects. For combination charts, the series is a chart, and you can also specify which chart to display.

Figure 23-25 shows a bubble, scatter, line, and combination chart configured with customized series. The charts illustrate how you might customize series colors, lines, and markers. The combination chart also shows how you might configure the series type to display an area, bar, and line chart.

Figure 23-25 Chart Series Customization

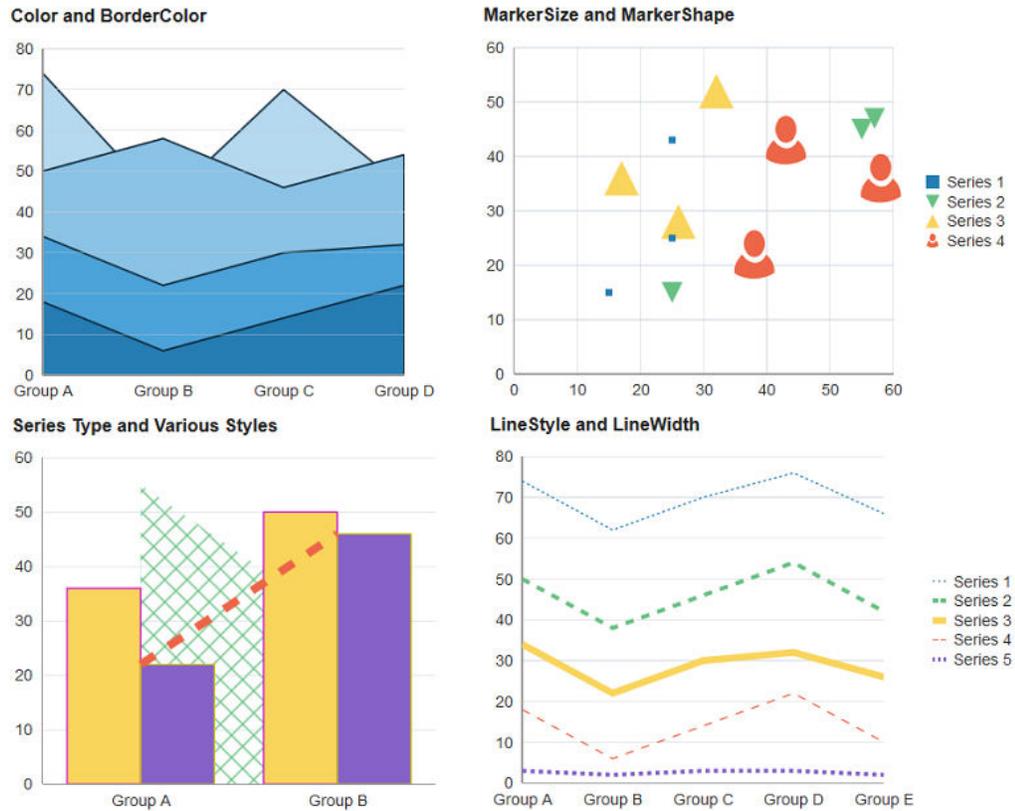


Chart Data Cursor

You can add a data cursor to a chart that the user can move to display detail about a data point.

Figure 23-26 shows a line chart configured with a data cursor. In this example, the user chose to display the detail for the chart's third series in the fourth group.

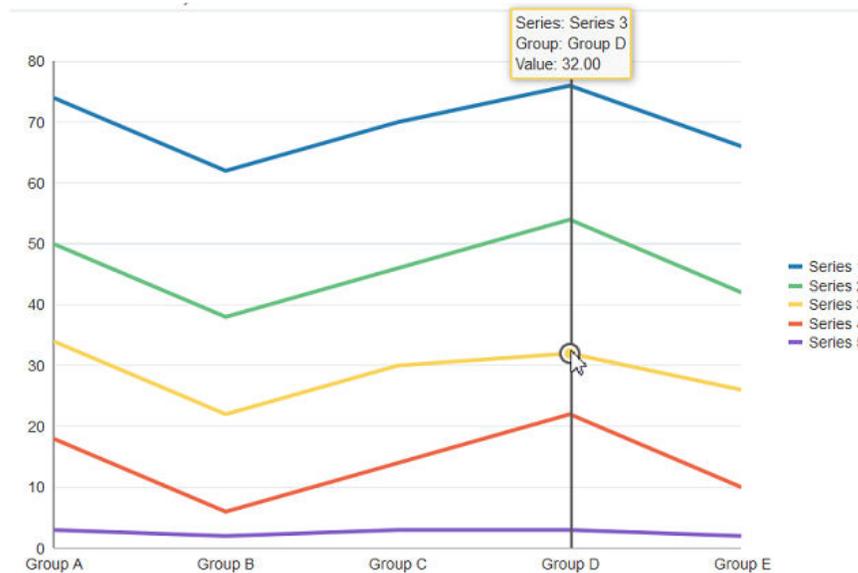
Figure 23-26 Line Chart Configured With Data Cursor

Chart Time Axis

Charts support the use of a time axis when the chart's data is based on dates. For example, you can use a time axis to display daily sales. The time data can cover regular or irregular time intervals. Time axes also support mixed frequency time data, where the time stamps vary by series.

[Figure 23-27](#) shows four charts configured with a time axis. The horizontal bar chart's time axis contains regular monthly data and is configured to show the year and month with nested labels. The irregular intervals bar chart shows the time axis configured for irregular yearly intervals. The `skipGaps` bar chart shows the time axis configured to remove the empty space for irregular yearly intervals. The combination chart shows a mixed frequency time axis with time data that varies by series.

Figure 23-27 Charts Configured With a Time Axis

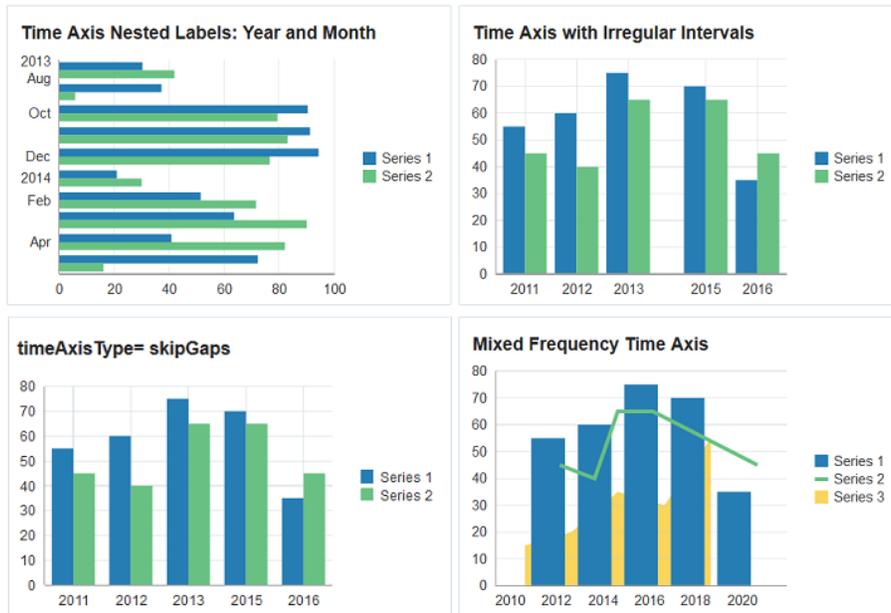


Chart Categorical Axis

You can configure a chart with a categorical axis, where groups are arranged in a drillable hierarchical structure. These groups can be enhanced with custom styles such as labels and tooltips. [Figure 23-28](#) shows a bar chart and an area chart with categorical axes. The area chart's axis has stylized data group labels and an uneven hierarchical distribution.

Figure 23-28 Hierarchical Data Groups and Labels

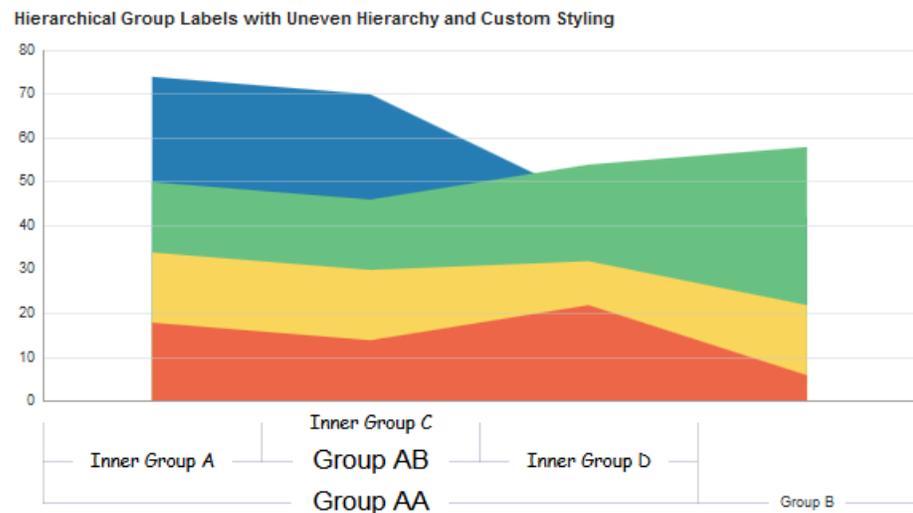
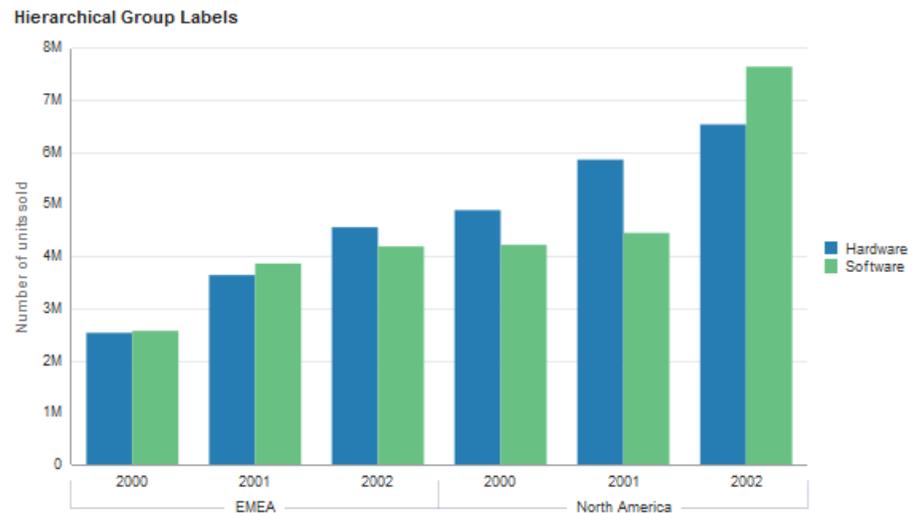


Chart Group Labels support custom CSS styling and tooltip support. [Figure 23-29](#) shows a bar chart where the group labels have different font and background colors, as well as a custom tooltip separate from the series tooltip that appears when hovering over a bar.

Figure 23-29 Custom Label Style and Tooltip

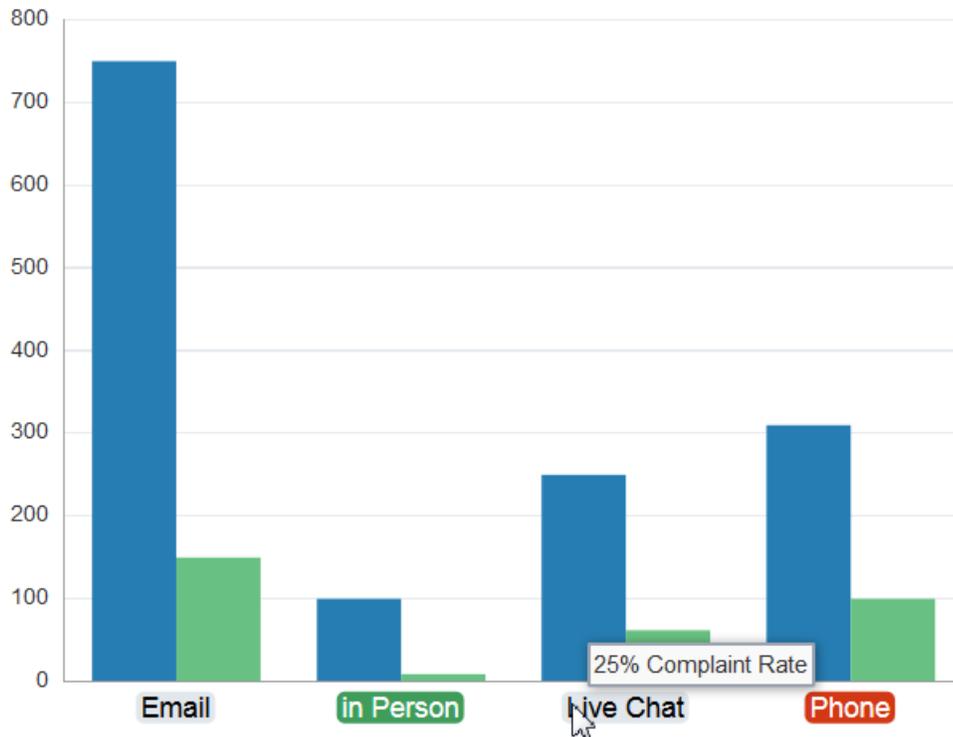


Chart Popups and Context Menus

You can configure charts to display popups or context menus using the `af:showPopupBehavior` tag.

Figure 23-30 shows a bar chart configured to show a popup when the user clicks the chart. The popup displays an output message in a note window.

Figure 23-30 Bar Chart Showing a Popup

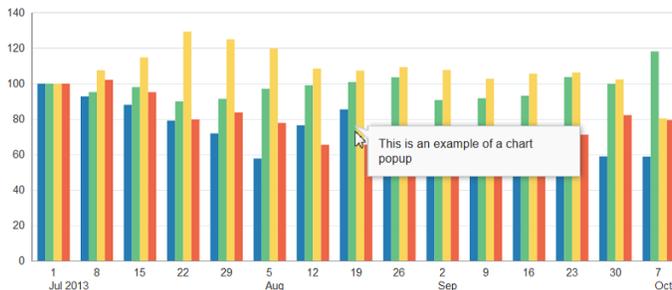


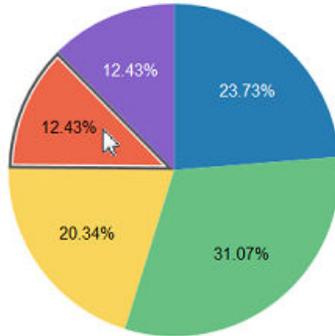
Chart Selection Support

Charts can be enabled for single or multiple selection of data markers such as bubbles in a bubble chart or shapes in a scatter chart. Enabling selection is required for popups

and context menus and for responding programmatically to user clicks on the data markers.

Figure 23-31 shows a pie chart enabled for multiple selection. Each data marker is highlighted as the user moves over it to provide a visual clue that the marker is selectable. The user can press **Ctrl** while selecting to add or delete slices from the selection.

Figure 23-31 Pie Chart Enabled for Selection



Pie charts in particular have multiple selection effects to better emphasize the selected slice.

Figure 23-32 shows pie charts with the selection effects highlight, explode, and highlight with explode.

Figure 23-32 Pie Chart Selection Effects

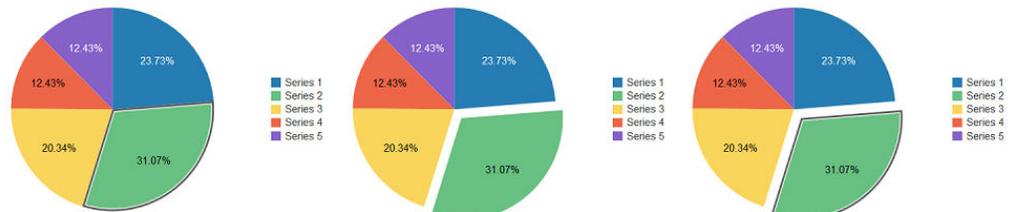
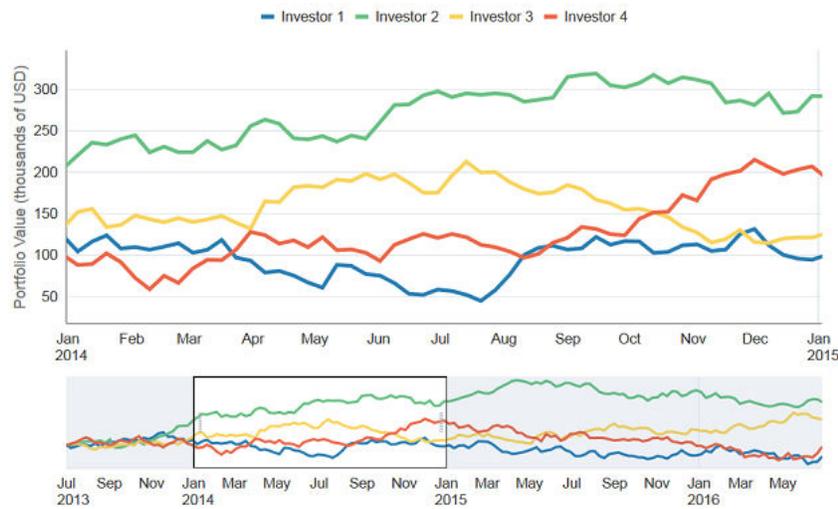


Chart Zoom and Scroll

Charts provide the ability to scroll through the data via a view port or simple scrollbar. This feature can be useful for charts containing large amounts of data.

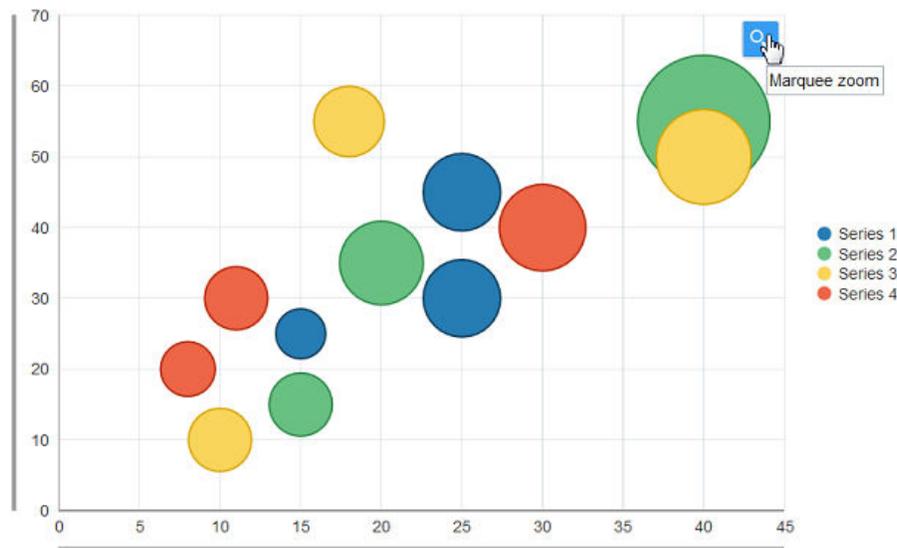
Figure 23-33 shows a line chart configured for scrolling with a view port. As the user moves the view port on the master line chart, the detail chart changes to reflect the selected range.

Figure 23-33 Line Chart Configured With Viewport for Scrolling



The user can also choose to zoom in on a specific period by clicking the **Marquee Zoom** icon that appears when the user hovers over the chart. [Figure 23-34](#) shows a bubble chart configured for marquee zoom. The user can drag the mouse over an area, and the chart zooms into the selected area.

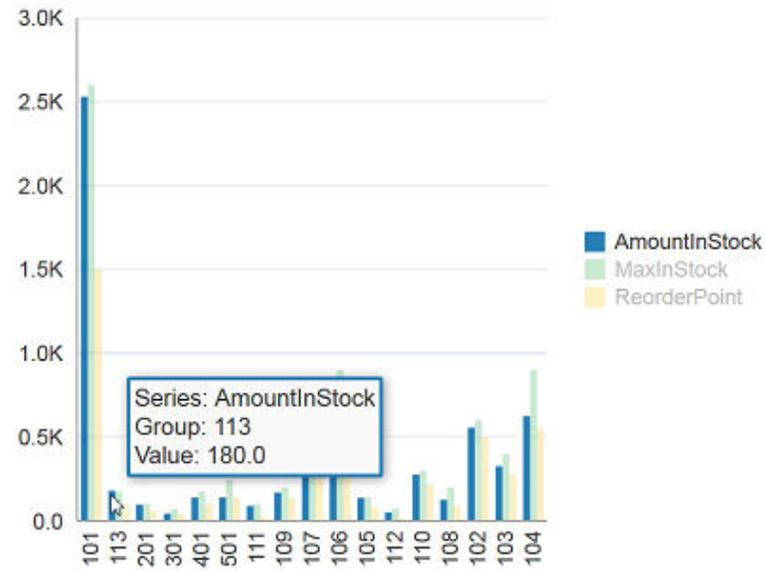
Figure 23-34 Bubble Chart Configured for Marquee Zoom



Legend and Marker Dimming

Charts provide the ability to highlight a series when the user hovers over a legend item or marker. [Figure 23-35](#) shows a bar chart configured with three series. As the user hovers over each series, the remaining series dim from view.

Figure 23-35 Chart Configured for Legend and Marker Dimming

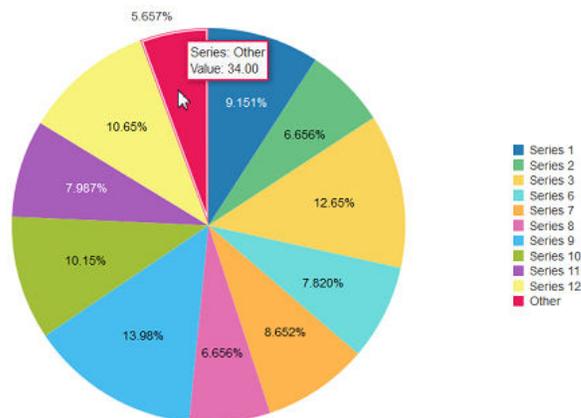


Pie Chart Other Slice Support

Pie charts provide the ability to aggregate data if your data model includes a large number of smaller contributors in relation to the larger contributors.

Figure 23-36 shows a pie chart configured to aggregate all values less than two percent of the total. In this example, the tooltip shows the total value of the aggregated slices.

Figure 23-36 Pie Chart Showing Other Slice Support

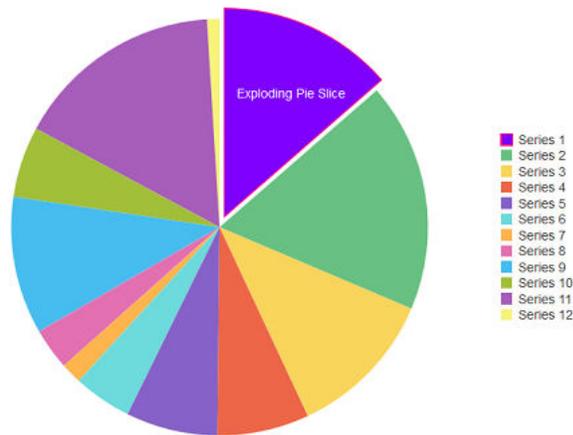


Exploding Slices in Pie Charts

When one slice is separated from the other slices in a pie, this display is referred to as an exploding pie slice. You can explode a slice to make it stand out from the other slices.

Figure 23-37 shows an example of an exploded pie slice.

Figure 23-37 Pie Chart Configured With Exploding Pie Slice



Active Data Support (ADS)

Charts support ADS by sending a Partial Page Refresh (PPR) request when an active data event is received. The PPR response updates the components, animating the changes as needed. Supported ADS events include:

- Update
- Remove
- Insert
- Refresh

The following limitations apply:

- EL Operators are not supported within the EL Expressions for active attributes.
- To insert a chart data item, you must include the series or group in the active data entry.
- You can't change the series or group of an existing item.
- The `dvt:chartSeriesStyle` tag is not supported on an inserted data item.
- Attribute groups are not supported.

For additional information about using the Active Data Service, see [Using the Active Data Service with an Asynchronous Backend](#).

Chart Animation

Charts support animation upon initial display or data change.

Chart Image Formats

Chart components rely on HTML5 technologies available in modern browsers for animations and interactivity.

Additional Functionality for Chart Components

You may find it helpful to understand other ADF Faces features before you implement your chart. Additionally, once you have added a chart to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that chart components can use:

- Client-side framework: DVT components are rendered on the client when the browser supports it. You can respond to events on the client using the `af:clientListener` tag. For more information, see [Listening for Client Events](#).
- Partial page rendering: You may want a chart to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: When enabled, users can change the way the chart displays at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: You can make your chart components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Skins and styles: You can customize the appearance of chart components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound charts based on how your ADF Business Components are configured. For more information, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Chart Component

To use the ADF DVT Chart component, add the selected chart type to a page using the Component Palette window. Then define the data for the chart and complete the additional configuration in JDeveloper using the tag attributes in the Properties window. You can easily configure a custom chart with many pre-built attributes..

Chart Component Data Requirements

The chart component use a standard `CollectionModel` for its data structure. This class extends the JSF `DataModel` class and adds on support for row keys and sorting. In the `DataModel` class, rows are identified entirely by index. This can cause problems when the underlying data changes from one request to the next, for example a user request to delete one row may delete a different row when another user adds a row. To work around this, the `CollectionModel` class is based on row keys instead of indexes. For more information about collection-based components, see [Using Tables, Trees, and Other Collection-Based Components](#).

Data requirements for charts differ with chart type. Data requirements can be any of the following kinds:

- **Geometric:** Some chart types need a certain number of data points in order to display data. For example, a line chart requires at least two groups of data because a line requires at least two points.
- **Complex:** Some chart types require more than one data point for each marker (which is the component that actually represents the data in a chart). A scatter chart, for example, needs two values for each group so that it can position the marker along the x-axis and along the y-axis. If the data that you provide to a chart does not have enough data points for each group, the chart component does its best to display a chart.
- **Logical:** Some chart types cannot accept certain kinds of data. The following examples apply:
 - **Negative data:** Do not pass negative data to a pie chart or a funnel chart.
 - **Null or zero data:** A bar or marker will not be visible if its value is zero, but an invisible region will be drawn to provide tooltip information.

- Insufficient sets (or series) of data: Dual-Y charts require a set of data for each y-axis. Usually, each set represents different information. For example, the y-axis might represent sales for specific countries and time periods, while the y2-axis might represent total sales for all countries. If you pass only one set of y-axis data, then the chart cannot display data on two different y-axes. It displays the data on a single y-axis.

Area, Bar, and Line Chart Data Requirements

Data requirements for area, bar, and line charts include:

- At least two groups of data are required for area and line charts. A group is represented by a position along the horizontal axis of area, bar, and line charts or vertical axis if the charts have horizontal orientations. In a chart that shows data for a three-month period, the groups might be labeled January, February, and March.
- One or more series of data is required. In a chart that shows data for a three-month period, the series might be sales and quota.
- Dual-Y charts require two sets of data.

Bar charts support an optional third value to specify bar widths.

Bubble Chart Data Requirements

Bubble charts require at least three data values for a data marker. Each data marker in a bubble chart represents three group values:

- The x value that determines the marker's location along the x-axis.
- The y value that determines the marker's location along the y-axis.
- The z value that determines the size of the marker.

For more than one group of data, bubble charts require that data be in multiples of three. For example, in a specific bubble chart, you might need three values for Paris, three for Tokyo, and so on. An example of these three values might be: x value is average life expectancy, y value is average income, and z value is population.

Note:

When you look at a bubble chart, you can identify groups of data by examining tooltips on the markers. However, identifying groups is not as important as looking at the overall pattern of the data markers.

Combination Chart Data Requirements

Combination charts require one set of data for each chart included in the combination chart. Each chart in the combination chart must meet the data requirements for the area, bar, or line chart components on which it is based. For a list of the data requirements for area, bar, or line chart components, see [Area, Bar, and Line Chart Data Requirements](#).

Funnel Chart Data Requirements

Data requirements for a funnel chart are:

- One set of values is required to represent the actual data items or slices in a funnel chart. In a chart that shows where customers get stuck in a process, the actual value would be the actual number of customers experiencing difficulty within a given time period.
- One stage value is required to represent the different slices. In a chart that shows where customers get stuck in a process, the stage value is the different steps in the process.
- One optional set of values may be provided to represent the target data items for each slice. In a chart that shows where customers get stuck in a process, the target value would be the maximum number of customers tolerable before the process needs to be updated.

Pie Chart Data Requirements

One collection of data with one or more sets of data items is required for a pie chart. The data structure is as follows:

- A series or set of data is represented by the pie slice. You see legend text for each set of this data. For example, if there is a separate set of data for each country, then the name of each country appears in the legend text.
- Data values cannot be negative.

Scatter Chart Data Requirements

Scatter charts require at least two data values for each marker. Each data marker represents the following:

- The x value that determines the marker's location along the x-axis.
- The y value that determines the marker's location along the y-axis.

For more than one group of data, the data must be in multiples of two.

Spark Chart Data Requirements

Line, bar, and area spark charts require a single series of data values.

Floating bar spark charts require two series of data values, one for the float offset, and one for the bar value.

Stock Chart Data Requirements

Stock charts require values to represent four attributes for each data marker. Data requirements for a stock chart are:

- Two set of values are required to represent the opening and closing values of stocks.
- For high-low stock charts, two additional sets of values are required to represent the high and low values of stocks for a given time period.

- If the type of chart includes volume, one set of data is required to represent the volume of trade for each stock for a given time period.

Configuring Charts

Because of the many chart types and the significant flexibility of the chart components, charts have a large number of DVT tags. The prefix `dvt:` occurs at the beginning of each chart tag name indicating that the tag belongs to the ADF Data Visualization Tools (DVT) tag library. The following list identifies groups of tags related to the chart component:

- Chart component tags: The nine chart component tags provide a convenient and quick way to create a chart type. They are represented in the Components window as categories of charts with one or more type variations.

[Table 23-1](#) provides a description of the chart component tags and their variations.

Table 23-1 Chart Component Tags

Chart Tag	Description	Variations
<code>areaChart</code>	Represents data as a filled in area.	dual y-axis stacked range
<code>barChart</code>	Represents data as a series of vertical bars.	dual y-axis stacked range
<code>bubbleChart</code>	Represents data by the location and size of the round (bubble) data marker.	
<code>comboChart</code>	Represents data as a combination of area, bar, or line markers.	dual y-axis stacked
<code>funnelChart</code>	Represents data as a stepped cone.	
<code>lineChart</code>	Represents data as a series of lines.	dual y-axis stacked
<code>pieChart</code>	Displays values that are parts of a whole, where each value is shown as a sector of a circle.	donut
<code>scatterChart</code>	Represents data by the location of data markers on a two-dimensional plane.	
<code>sparkChart</code>	Simple, condensed chart that displays trends or variations in a single data value, typically stamped in the column of a table or in line with related text.	area bar floating bar line

Table 23-1 (Cont.) Chart Component Tags

Chart Tag	Description	Variations
stockChart	Displays stock information and volume of trade for a given time period.	Open-Close Open-Close with Volume Open-High-Low-Close Open-High-Low-Close with Volume

- Chart child component tags: These child tags are supported by most chart components to provide a variety of customization options.

[Table 23-2](#) provides a list and description of these child tags.

Table 23-2 Common Chart Child Tags

Child Tag	Description
chartDataItem	Defines properties for the data item of an area, bar, bubble, combination, line, or scatter chart.
chartGroup	Defines properties for the group of an area, bar, bubble, combination, line, or scatter chart.
chartLegend	Defines properties for the chart legend.
chartSeriesStyle	Defines properties for the series of an area, bar, bubble, combination, line, or scatter chart.
chartValueFormat	Defines formatting properties for the values of a chart.
chartXAxis	Defines properties for the x-axis of a chart.
chartYAxis	Defines properties for the y-axis of a chart.
chartY2Axis	Defines properties for the y2-axis of a chart.
chartAxisLine	Child tags of the chartXAxis, chartYAxis, and chartY2Axis components. Provides additional customization for the chart axes.
chartTickLabel	
majorTick	
minorTick	
referenceArea	
referenceLine	
pieDataItem	Defines properties for the data item of a pie chart.
funnelDataItem	Defines properties for the data item of a funnel chart.
stockDataItem	Defines properties for the data item of a funnel chart.

- Chart attributes: Properties of chart components. Attributes may be applicable to all charts or specific to a chart type.

[Table 23-3](#) provides a list and description of commonly used attributes.

Table 23-3 Common Chart Attributes

Child Tag	Description
animationIndicators	Specifies the type of data change animation indicator.

Table 23-3 (Cont.) Common Chart Attributes

Child Tag	Description
barGapRatio	Specifies the amount of space between bars in a bar chart and the thickness of the bars, expressed as a ratio or percentage.
centerLabel	Specifies an inner string and its font for a ring chart (pie chart).
centerLabelStyle	
coordinateSystem	Specifies if the chart uses Cartesian or Polar coordinates.
dataCursor	Specifies whether or not the data cursor is enabled.
dataLabelPosition	Specifies the position of the data labels. Options depend on the chart type.
dataSelection	Specifies the selection mode for the chart.
drilling	Specifies whether this group can be drilled. Valid values are 'off' (default) and 'on'. When set to 'on', the existing drillEvent will be fired when the user clicks/taps on a the group label.
footnote	Defines the footnote and its horizontal alignment.
footnoteHAlign	
hideAndShowBehavior	Specifies the hide and show behavior when clicking on legend items.
high	Specifies the upper limit of range data for range area and range bar charts.
hoverBehavior	Specifies whether or not to dim other markers when the user hovers over a marker.
initialZoom	Specifies whether or not to initially zoom into the first or last portion of a chart.
innerRadius	Specifies a ratio or percentage of an inner circle in a pie chart to be hidden, turning it into a ring chart.
labelPosition	Specifies the position of data labels. Options depend on the chart type.
low	Specifies the lower limit of range data for range area and range bar charts.
maximumBarWidth	Specifies the maximum width of bars in a bar chart, expressed in pixels.
orientation	Specifies whether the chart is displayed vertically or horizontally.
otherColor	Defines the color and percentage for the Other slice in a pie chart.
otherThreshold	
polarGridShape	Specifies if the shape of the polar grid is circle or polygon.
selectionEffect	Specifies the selection effect when a pie slice is selected in a pie chart.
seriesEffect	Defines the fill properties for data items.
sliceGaps	Specifies the amount of gap in between pie slices in a pie chart and in between funnel segments in a funnel chart. Accepted values lie between 0 and 1.
sliceLabelPosition	Specifies the position of the pie chart's data labels.

Table 23-3 (Cont.) Common Chart Attributes

Child Tag	Description
sorting	Specifies if pie chart and bar chart data items should be sorted. On bar charts, accepted values are ascending, descending and off.
splitDualY	Specifies whether to split information that is provided using the y2-axis. When this attribute is set to 'on' the chart will render the two data sets separately in stacked plot areas that share the same x-axis.
splitterPosition	Specifies a value that will correspond to how much of the available plot area the y-axis will occupy. This attribute applies only when the <code>splitDualY</code> attribute is set to 'on'. Valid values range from 0 to 1. The default value is .5.
stack	Specifies whether or not the data should be stacked.
subtitle	Specifies the subtitle for the chart.
timeAxisType	Defines a time axis type for the chart.
symbolWidth	Specifies the common size attributes for all symbols in a chart legend.
symbolHeight	
title	Defines the title and its horizontal alignment.
titleHAlign	
var	EL variable that iterates through each element in the collection.
zoomAndScroll	Specifies the chart's zoom and scroll behavior.

- Chart facets: All charts with the exception of spark charts, stock charts, funnel charts, and pie charts support facets, which are named sections within a component.

[Table 23-4](#) provides a list and description of supported chart facets.

Table 23-4 Chart Facets

Child Tag	Description
dataStamp	Wraps the data item component or components to stamp for each row of the model. To stamp multiple data items, wrap them in the <code>af:group</code> tag.
groupStamp	Specifies the group component to stamp for each column of the model. The properties of the stamp will be processed once for each unique group. To declaratively define multiple groups, wrap them in the <code>af:group</code> tag.
overview	Specifies the rendering of the optional overview window.
seriesStamp	Specifies the series style component to stamp for each row of the model. The properties of the stamp will be processed once for each unique series. To declaratively define multiple series, wrap them in the <code>af:group</code> tag.

- Spark chart tags: Properties of the spark chart component. Spark charts contain a minimal set of formatting attributes.

Table 23-5 provides a list and description of commonly used spark chart tags.

Table 23-5 Common Spark Chart Attributes

Child Tag	Description
<code>axisScaledFromBaseline</code>	Specifies whether or not the axis is scaled to include the baseline value of zero.
<code>borderColor</code>	Specifies the border color of the data item.
<code>color</code>	Specifies the color of the bars, line, or area in the spark chart.
<code>lineWidth</code>	Specifies line width in pixels, whether the line is solid, dashed or dotted, and the type of line connector, such as straight, curved, stepped or segmented.
<code>lineStyle</code>	
<code>lineType</code>	
<code>firstMarkerColor</code>	Specifies the colors for the first, last, high, and low markers and whether or not the markers are displayed.
<code>highMarkerColor</code>	
<code>lastMarkerColor</code>	
<code>lowMarkerColor</code>	
<code>markers</code>	
<code>markerShape</code>	Specifies the shape of the marker, such as <code>square</code> , <code>circle</code> , <code>diamond</code> , <code>plus</code> , <code>triangleUp</code> , and <code>triangleDown</code> . Also specifies the size of the marker in pixels.
<code>markerSize</code>	
<code>sparkItem</code>	Specifies the data value for the spark chart
<code>subType</code>	Specifies whether the spark chart is displayed as an area, bar, floating bar, or line.
<code>dataSelection</code>	Specifies the selection mode for the chart.
<code>threshold</code>	Defines thresholds for the spark chart.
<code>thresholdSet</code>	
<code>tooltip</code>	Specifies text to display when the user hovers over the spark chart.
<code>timeAxisType</code>	Defines a time axis type for the chart.

For complete descriptions of all the tags, their attributes, and a list of valid values, consult the DVT tag documentation. To access this documentation for a specific chart tag in JDeveloper, select the tag in the Structure window and press F1 or click **Component Help** in the Properties window.

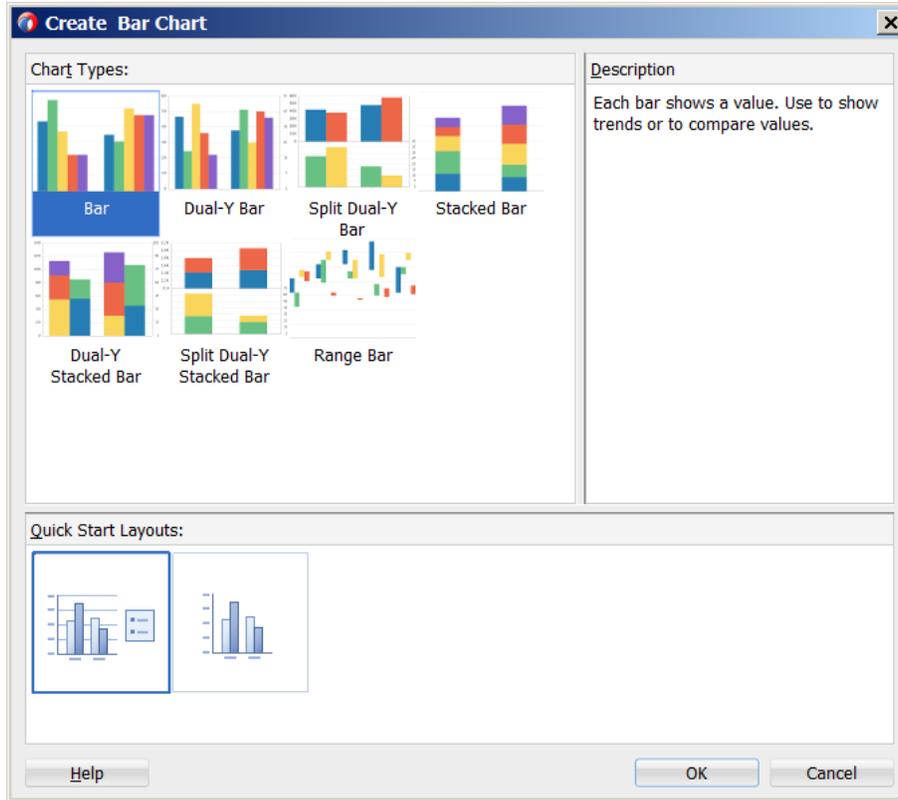
DVT charts also share many of the same attributes as other ADF Faces and DVT components. For additional information, see [Additional Functionality for Chart Components](#).

How to Add a Chart to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a chart to a JSF page. When you drag and drop a chart component onto the page, a Create Chart dialog displays available categories of chart types, with descriptions, to provide visual assistance when creating charts. You can also specify a quick start layout of the chart's legend.

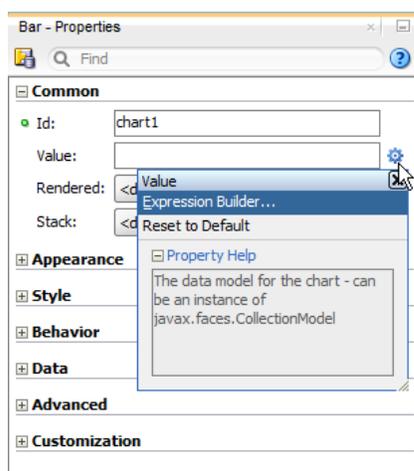
Figure 23-38 shows the Create Bar Chart dialog for bar charts with the default bar chart type and quick start layout selected.

Figure 23-38 Create Bar Chart Dialog



Once you complete the dialog, and the chart is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the chart.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. Figure 23-39 shows the dropdown menu for a bar chart component `value` attribute.

Figure 23-39 Bar Chart Value Attribute**Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a chart and the binding will be done for you. For more information, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*

Before you begin:

It may be helpful to have an understanding of how chart attributes and chart child tags can affect functionality. For more information, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a chart to a page:

1. In the ADF Data Visualizations page of the Components window, from the Charts panel, drag and drop the desired chart category onto the page to open the Create Chart dialog.
2. Use the dialog to select the chart type and the quick start layout for display of chart title, legend, and labels. For help with the dialog, press F1 or click **Help**.
3. Click **OK** to add the chart to the page.
4. In the Properties window, view the attributes for the chart. Use the **Component Help** button to display the complete tag documentation for the chart.

What Happens When You Add a Chart to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a chart from the Components window onto a JSF page.

The example below shows the code inserted in the JSF page for a bar chart with the quick start layout selected in [Figure 23-38](#).

```
<dvt:barChart id="chart1">
  <dvt:chartLegend id="leg1" rendered="true"/>
</dvt:barChart>
```

After inserting a chart component into the page, you can use the visual editor or Properties window to add data or customize chart features. For information about setting component attributes, see [How to Set Component Attributes](#).

Adding Data to Charts

You can configure ADF DVT chart components to render data by specifying the required data model or method in the chart's value attribute and then configuring an appropriate data item child component to interpret the data values.

The process to add data to charts depends upon the chart's type. In most cases, you specify the data model in the chart's value attribute and configure the chart's data items in the `chartDataItem` child component. For pie, funnel, and stock charts, you specify the value in the `pieDataItem`, `funnelDataItem`, or `stockDataItem` child component respectively. For spark charts, you specify the spark chart's value in the `sparkItem` child component.

You can specify the chart's value and chart data items in a managed bean that returns the chart's data or by binding a data control to the chart.

How to Add Data to Area, Bar, Combination, and Line Charts

To add data to area, bar, combination, and line charts, specify the data model in the chart's value attribute and configure a `chartDataItem` for each unique group. For details about data requirements, see [Chart Component Data Requirements](#).

The code below shows an example of a managed bean that defines the data for a line chart that shows weekly portfolio values for four investors. In this example, the `ChartDataSource` class defines the chart's `CollectionModel` in the `getWeeklyStockData()` method which calls the `getPortfolioData()` method to define the chart's data items.

```
package view;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.HashMap;
import java.util.Random;

import java.util.List;

import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.ModelUtils;
```

```
public class ChartDataSource {
    /**
     * Object representing the data for a single row of the model.
     */
    public static class ChartDataItem extends HashMap<String, Object> {
        @SuppressWarnings("compatibility")
        private static final long serialVersionUID = 1L;

        public ChartDataItem(String series, Object group, Number value) {
            put("series", series);
            put("group", group);
            put("value", value);
        }

        public ChartDataItem(String series, Object group, Object x, Number y) {
            put("series", series);
            put("group", group);
            put("x", x);
            put("y", y);
        }

        public ChartDataItem(String series, Object group, Number x, Number y, Number z) {
            put("series", series);
            put("group", group);
            put("x", x);
            put("y", y);
            put("z", z);
        }
    }

    public CollectionModel getWeeklyStockData() {
        return getPortfolioData(4, 157, 2013, 6, 1, Calendar.DATE, 7);
    }

    private Random random = new Random(23);

    private double getNextValue(double curValue, double std) {
        if (curValue == 0)
            return 0;
        else
            return Math.max(curValue + random.nextGaussian() * std, 0);
    }

    public CollectionModel getPortfolioData(int numSeries, int numGroups,
                                           int startYear, int startMonth,
                                           int startDate, int dateField,
                                           int addCount)
    {
        List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
        GregorianCalendar cal;
        double curValue;
        for(int series=0; series<numSeries; series++) {
            cal = new GregorianCalendar(startYear, startMonth, startDate);
            curValue = 100;
            for(int group=0; group<numGroups; group++) {
                dataItems.add(new ChartDataItem("Investor " + (series+1),
                                                cal.getTime(),
                                                curValue));
                cal.add(dateField, addCount);
                curValue = getNextValue(curValue, 10);
            }
        }
    }
}
```

```

    }
  }
  return ModelUtils.toCollectionModel(dataItems);
}
}

```

The example below shows the code on the JSF page that defines the line chart and references the `ChartDataSource` class. In this example, the managed bean is named `chartDataSource`.

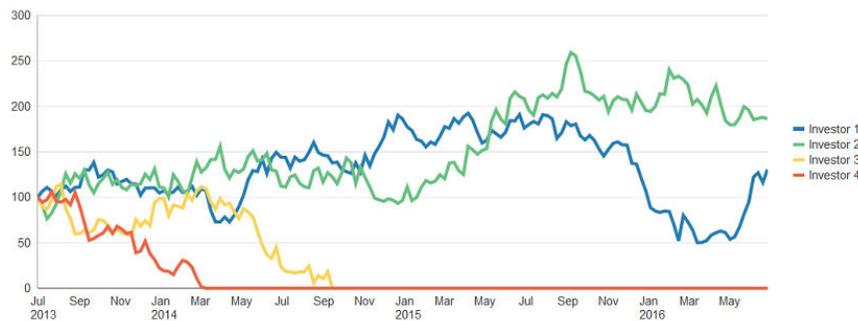
```

<dvt:lineChart id="chart1" value="#{chartDataSource.weeklyStockData}"
    timeAxisType="enabled" var="row">
  <dvt:chartLegend id="leg1" rendered="true" position="top"/>
  <f:facet name="dataStamp">
    <dvt:chartDataItem id="cdil" series="#{row.series}" group="#{row.group}"
      value="#{row.value}"/>
  </f:facet>
</dvt:lineChart>

```

Figure 23-40 shows the line chart that is displayed if you configure the line chart using the `ChartDataSource` class.

Figure 23-40 Line Chart Showing Portfolio Data for Four Investors



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add an area, bar, combination, or line chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to an area, bar, combination, or line chart:

1. Optionally, create the managed bean that will return the chart's data model.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).

2. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
3. In the Properties window, expand the **Data** section, and enter a value for the **Var** field.

The `var` property is the name of an EL variable that references each element in the data collection. In the second code example above, the `var` attribute is defined as `row`. The relevant code is shown below.

```
<dvt:lineChart id="chart1" value="#{chartDataSource.weeklyStockData}"
              timeAxisType="enabled" var="row">
```

4. In the Properties window, do one of the following:
 - To reference a managed bean that returns the chart's data model, expand the **Common** section and specify the EL expression that references the data model in the **Value** field.

You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `chartDataSource` that includes the class referenced in the first code sample above, enter the following in the **Value** field:

```
#{chartDataSource.weeklyStockData}.
```

For help with creating EL expressions, see [How to Create an EL Expression](#).

- To bind the chart to a data control, click **Bind to ADF Control** to select a data collection.
- For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
5. If you specified a managed bean for the chart's data model, do the following to add a `chartDataItem` to the chart:

- a. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Facet dataStamp**.
- b. To add multiple chart data items, right-click **f:facet - dataStamp** and choose **Insert Inside Facet data Stamp > Group**.
- c. Right-click **f:facet - dataStamp** or **af:group** and choose **Insert Inside (Facet data Stamp or Group) > Chart Data Item**.
- d. To add additional data items, right-click **af:group** and choose **Insert Inside Group > Chart Data Item** for each additional data item.
- e. Right-click **dvt:chartDataItem** and choose **Go to Properties**.
- f. In the Properties window, in the **Common** section, enter a value for the **Value** field. If creating a range area or range bar chart, enter values for the **High** and **Low** fields instead.

For example, to reference the value defined in the `chartDataSource` managed bean, enter the following in the **Value** field: `#{row.value}`.

In this example, `row` is the variable that you defined in the previous step, and `value` is defined in the `chartDataSource` bean's `getPortfolioData()` method shown in the first code sample above.

- g. Expand the **Data** section, and enter a value in the **Group** field.

To use the same `chartDataSource` managed bean, enter the following in the **Group** field: `#{row.group}`.

- h. Enter a value in the **Series** field.

To use the same `chartDataSource` managed bean, enter the following in the **Series** field: `#{row.series}`.

- i. To customize the tooltip that is displayed when the user moves the focus to a data point, enter a value in the `shortDesc` field.
- j. Configure any additional properties as needed to customize the data item.

For example, you can add a label or configure a series marker. For additional information, see [Customizing Chart Display Elements](#).

If your chart includes time data on the x-axis, you can enable a time axis. For additional information, see [How to Configure a Time Axis](#).

If your bar chart has discrete data points of varying weight or importance, you can enter a value in the **Z** field in the **Data** section. This value will be used to change the width of the bars.

If your area or bar chart has range data, you can enter values in the **High** and **Low** fields instead of **Value**. These values will be used to represent the upper and lower limits of each data point.

- k. To configure additional chart data items, repeat Step f through Step i for each additional data item.

How to Add Data to Pie Charts

To add data to a pie chart, specify the data model in the pie chart's `value` attribute and configure a `pieDataItem` for each unique series. For information about pie chart data requirements, see [Pie Chart Data Requirements](#).

The process to add data to pie charts is similar to the process for adding data to area, bar, bubble, combination, line, and scatter charts. The primary difference is that you add a `pieDataItem` instead of a `chartDataItem` to the chart.

The code below shows a simple example for configuring a pie chart from a managed bean. In this example, the `getDefaultPieData()` method is added to the `chartDataSource` bean referenced in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

```
public CollectionModel getDefaultPieData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    dataItems.add(new ChartDataItem("Series 1", "Group A", 42));
    dataItems.add(new ChartDataItem("Series 2", "Group A", 55));
    dataItems.add(new ChartDataItem("Series 3", "Group A", 36));
    dataItems.add(new ChartDataItem("Series 4", "Group A", 22));
    dataItems.add(new ChartDataItem("Series 5", "Group A", 22));
    return ModelUtils.toCollectionModel(dataItems);
}
```

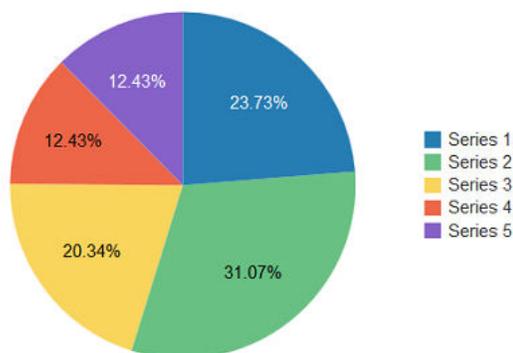
The example below shows the code on the JSF page that defines a pie chart and references the `ChartDataSource` class. In this example, the managed bean is named `chartDataSource`.

```
<dvt:pieChart id="chart1" value="#{chartDataSource.defaultPieData}" var="row">
  <dvt:chartLegend id="leg1" rendered="true"/>
</dvt:pieChart>
```

```
<dvt:pieDataItem label="#{row.series}" value="#{row.value}" id="pdil"/>
</dvt:pieChart>
```

Figure 23-41 shows the pie chart at run time. In this example, the pie chart is rendered with a legend, and the data labels show the percentage of each slice as a portion of the total.

Figure 23-41 Pie Chart Configured From Managed Bean



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a pie chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to a pie chart:

1. Optionally, create the managed bean that will return the chart's data model.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
2. In the Structure window, right-click **dvt:pieChart** and choose **Go To Properties**.
3. In the Properties window, do one of the following:
 - To reference a managed bean that returns the chart's data model, expand the **Common** section and specify the EL expression that references the data model in the **Value** field.

You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `chartDataSource` that includes the method to supply data to a pie chart, as shown in the first code sample in [How to Add Data to Area, Bar, Combination,](#)

and [Line Charts](#), enter the following in the **Value** field:

```
#{chartDataSource.defaultPieData}.
```

For help with creating EL expressions, see [How to Create an EL Expression](#).

- To bind the chart to a data control, click **Bind to ADF Control** to select a data collection.

For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

4. If you specified a managed bean for the chart's data model, do the following to add a `pieDataItem` to the pie chart:
 - a. Right-click **dvt:pieChart** and choose **Insert Inside Pie > Pie Data Item**.
 - b. In the Insert Pie Data Item dialog, enter a label and value for the data item.

For example, to use the `chartDataSource` managed bean, enter the following for the label and value, respectively:

```
#{row.series}  
#{row.value}
```

- c. In the Structure window, right-click **dvt:pieChart** and choose **Go to Properties**.
- d. In the Properties window, expand the **Data** section, and enter a value for the **Var** field.

The `var` property is the name of an EL variable that references each element in the data collection. In the second code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#), the `var` attribute is defined as `row`. The relevant code is shown below.

```
<dvt:lineChart id="chart1" value="#{chartDataSource.weeklyStockData}"  
    timeAxisType="enabled" var="row">
```

- e. To customize the tooltip that is displayed when the user moves the focus to a data point, enter a value in the **shortDesc** field.
- f. To convert the pie chart into a ring chart, enter a value in the following fields:
 - **InnerRadius**: Enter a value between 0 and 1 to determine the size of the ring hole.
 - **CenterLabel**: Enter a string to add a chart label in the center of the ring.
 - **CenterLabelStyle**: Enter values to style and format the center label.
 - **SliceGaps**: Enter a value between 0 and 1 to determine the size of gaps between individual pie slices.
- g. Configure any additional properties as needed to customize the data item.
For example, you can customize the label or explode the pie slice. For additional information, see [Customizing Chart Display Elements](#).
- h. To add additional pie data items, repeat Step a through Step e for each additional data item.

How to Add Data to Bubble or Scatter Charts

To add data to a bubble or scatter chart, specify the data model in the chart's `value` attribute and configure a `chartDataItem` for each unique group. For details about data requirements, see [Chart Component Data Requirements](#).

The process to add data to bubble or scatter charts is similar to the process for adding data to area, bar, bubble, combination, line, and scatter charts. The primary difference is that you must also specify values for the x-axis and y-axis. For bubble charts, you must also specify values for the z-axis, which represents the size of the bubbles.

The example below shows a simple example for configuring a bubble chart from a managed bean. In this example, the `getDefaultBubbleData()` method is added to the `chartDataSource` bean referenced in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

```
public CollectionModel getDefaultBubbleData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    // Each data item below defines a series, group, x, y, and z value
    dataItems.add(new ChartDataItem("Series 1", "Group A", 15, 25, 5));
    dataItems.add(new ChartDataItem("Series 1", "Group B", 25, 30, 12));
    dataItems.add(new ChartDataItem("Series 1", "Group C", 25, 45, 12));

    dataItems.add(new ChartDataItem("Series 2", "Group A", 15, 15, 8));
    dataItems.add(new ChartDataItem("Series 2", "Group B", 20, 35, 14));
    dataItems.add(new ChartDataItem("Series 2", "Group C", 40, 55, 35));

    dataItems.add(new ChartDataItem("Series 3", "Group A", 10, 10, 8));
    dataItems.add(new ChartDataItem("Series 3", "Group B", 18, 55, 10));
    dataItems.add(new ChartDataItem("Series 3", "Group C", 40, 50, 18));

    dataItems.add(new ChartDataItem("Series 4", "Group A", 8, 20, 6));
    dataItems.add(new ChartDataItem("Series 4", "Group B", 11, 30, 8));
    dataItems.add(new ChartDataItem("Series 4", "Group C", 30, 40, 15));
    return ModelUtils.toCollectionModel(dataItems);
}
```

For scatter charts, only the x-axis and y-axis positions are required. The example below shows a simple example for configuring a scatter chart from a managed bean. In this example, the `getDefaultScatterData()` method is added to the `chartDataSource` bean referenced in the first code sample shown in [How to Add Data to Area, Bar, Combination, and Line Charts](#)

```
public CollectionModel getDefaultScatterData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    dataItems.add(new ChartDataItem("Series 1", "Group A", 15, 15));
    dataItems.add(new ChartDataItem("Series 1", "Group B", 25, 43));
    dataItems.add(new ChartDataItem("Series 1", "Group C", 25, 25));

    dataItems.add(new ChartDataItem("Series 2", "Group A", 25, 15));
    dataItems.add(new ChartDataItem("Series 2", "Group B", 55, 45));
    dataItems.add(new ChartDataItem("Series 2", "Group C", 57, 47));

    dataItems.add(new ChartDataItem("Series 3", "Group A", 17, 36));
    dataItems.add(new ChartDataItem("Series 3", "Group B", 32, 52));
    dataItems.add(new ChartDataItem("Series 3", "Group C", 26, 28));

    dataItems.add(new ChartDataItem("Series 4", "Group A", 38, 22));
}
```

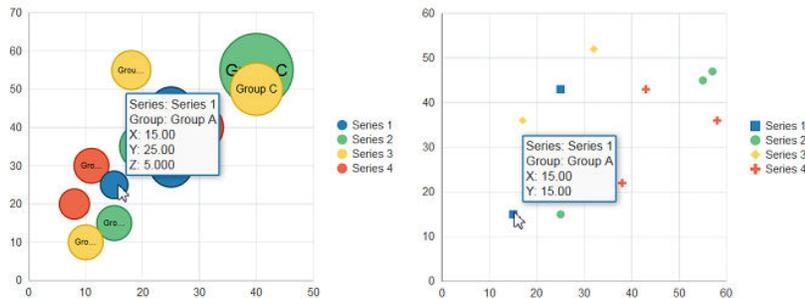
```

dataItems.add(new ChartDataItem("Series 4", "Group B", 43, 43));
dataItems.add(new ChartDataItem("Series 4", "Group C", 58, 36));
return ModelUtils.toCollectionModel(dataItems);
}

```

Figure 23-42 shows the bubble chart and scatter chart displayed at runtime. The tooltips show the data from the first `ChartDataItem` defined in the code samples to supply data to bubble and scatter charts, as shown in [How to Add Data to Bubble or Scatter Charts](#).

Figure 23-42 Bubble and Scatter Charts Configured From Managed Bean



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a bubble or scatter chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to a bubble chart:

1. Optionally, create the managed bean that will return the chart's data model.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
2. In the Structure window, right-click `dvt:typeChart` and choose **Go To Properties**.
3. In the Properties window, expand the **Data** section, and enter a value for the **Var** field.
The `var` property is the name of an EL variable that references each element in the data collection. For the bubble and scatter charts shown in [Figure 23-42](#), the `var` attribute is defined as `row`.
4. In the Properties window, do one of the following:
 - To reference a managed bean that returns the chart's data model, expand the **Common** section and specify the EL expression that references the data model in the **Value** field.

You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `chartDataSource` that includes the method referenced in the first code sample in [How to Add Data to Bubble or Scatter Charts](#), enter the following in the **Value** field: `#{chartDataSource.defaultBubbleData}`.

For help with creating EL expressions, see [How to Create an EL Expression](#).

- To bind the chart to a data control, click **Bind to ADF Control** to select a data collection.

For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

5. If you specified a managed bean for the chart's data model, do the following to add a `chartDataItem` to the chart:

- a. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Facet dataStamp**.
- b. To add multiple chart data items, right-click **f:facet - dataStamp** and choose **Insert Inside Facet data Stamp > Group**.
- c. Right-click **f:facet - dataStamp** or **af:group** and choose **Insert Inside (Facet data Stamp or Group) > Chart Data Item**.
- d. To add additional data items, right-click **af:group** and choose **Insert Inside Group > Chart Data Item** for each additional data item.
- e. Right-click **dvt:chartDataItem** and choose **Go to Properties**.
- f. Expand the **Data** section, and enter a value in the **X** field.

To use the managed beans shown in the code samples to supply data to bubble and scatter charts, as shown in [How to Add Data to Bubble or Scatter Charts](#), enter the following in the **X** field: `#{row.x}`.

- g. Enter a value in the **Y** field.

For example, enter the following in the **Y** field: `#{row.y}`.

- h. For bubble charts, enter a value in the **Z** field.

For example, enter the following in the **Z** field: `#{row.z}`.

- i. Enter a value in the **Group** field.

For example, enter the following in the **Group** field: `#{row.group}`.

- j. Enter a value in the **Series** field.

For example, enter the following in the **Series** field: `#{row.series}`.

- k. To customize the tooltip that is displayed when the user moves the focus to a data point, enter a value in the `shortDesc` field.

- l. Configure any additional properties as needed to customize the data item.

For example, you can add a label or configure a series marker. For additional information, see [Customizing Chart Display Elements](#).

- m. To configure additional chart data items, repeat Step 5 through Step 12 for each additional data item.

How to Add Data to Funnel Charts

To add data to a funnel chart, specify the data model in the chart's `value` attribute and configure a `funnelDataItem` for each unique group. For details about data requirements, see [Chart Component Data Requirements](#).

The process to add data to funnel charts is similar to the process for adding data to area, bar, horizontal bar, bubble, combination, line, and scatter charts. The primary difference is that you add a `funnelDataItem` instead of a `chartDataItem` to the chart.

The example below shows a simple example for configuring a funnel chart from a managed bean. In this example, the `getDefaultFunnelData()` method and the overloaded `ChartDataItem` constructor are added to the `chartDataSource` bean referenced in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

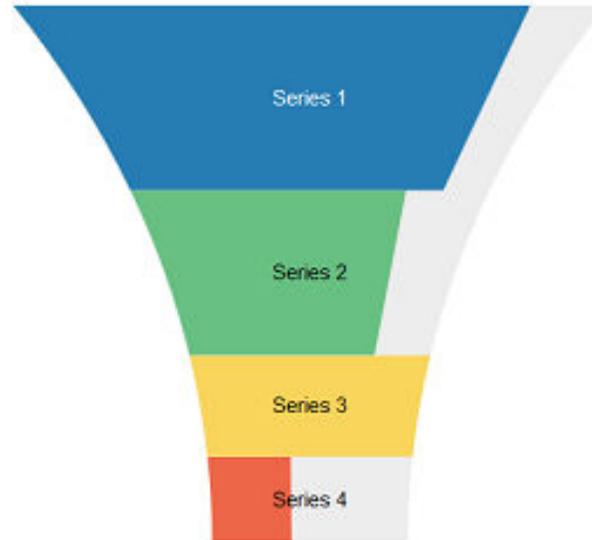
```
public ChartDataItem(String series, Number value, Number targetValue) {
    put("series", series);
    put("value", value);
    put("targetValue", targetValue);
}

public CollectionModel getDefaultFunnelData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    dataItems.add(new ChartDataItem("Series 1", 46, 60));
    dataItems.add(new ChartDataItem("Series 2", 34, 50));
    dataItems.add(new ChartDataItem("Series 3", 30, 30));
    dataItems.add(new ChartDataItem("Series 4", 11, 25));
    return ModelUtils.toCollectionModel(dataItems);
}
```

The example below shows the code on the JSF page that defines a funnel chart and references the `ChartDataSource` class. In this example, the managed bean is named `chartDataSource`.

```
<dvt:funnelChart id="chart1" value="#{chartDataSource.defaultFunnelData}"
var="row">
    <dvt:funnelDataItem label="#{row.series}" value="#{row.value1}"
targetValue="#{row.targetValue1}" id="pdil1"/>
</dvt:funnelChart>
```

[Figure 23-43](#) shows the funnel chart at run time. The target values specified are optional, and allow the generation of an incomplete funnel as seen below.

Figure 23-43 Funnel Chart Configured From Managed Bean

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

Add a funnel chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to a funnel chart:

1. Optionally, create the managed bean that will return the chart's data model.

If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).

2. In the Structure window, right-click **dvt:funnelChart** and choose **Go To Properties**.

3. In the Property Inspector, do one of the following:

- To reference a managed bean that returns the chart's data model, expand the **Common** section and specify the EL expression that references the data model in the **Value** field.

You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `chartDataSource` that includes the method shown in the first code sample above, enter the following in the **Value** field:

```
#{chartDataSource.defaultFunnelData}
```

For help with creating EL expressions, see [How to Create an EL Expression](#).

- To bind the chart to a data control, click **Bind to ADF Control** to select a data collection.

For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

4. If you specified a managed bean for the chart's data model, do the following to add a `funnelDataItem` to the funnel chart:
 - a. Right-click **dvt:funnelChart** and choose **Insert inside dvt:funnelChart > Funnel Data Item**.
 - b. In the Insert Funnel Data Item dialog, enter a label and value for the data item. For example, to use the `chartDataSource` managed bean, enter the following for the label and value, respectively:

```
#{row.series}
#{row.value}
```

- c. In the Structure window, right-click **dvt:funnelChart** and choose **Go to Properties**.
- d. In the Property Inspector, expand the **Data** section, and enter a value for the **Var** field.

The `var` property is the name of an EL variable that references each element in the data collection. In the second code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#), the `var` attribute is defined as `row`.
- e. To customize the tooltip that is displayed when the user moves the focus to a data point, enter a value in the `shortDesc` field.
- f. Configure any additional properties as needed to customize the data item. For example, you can customize the label. For additional information, see [Customizing Chart Display Elements](#).
- g. To add additional funnel data items, repeat Step 1 through Step 6 for each additional data item.

How to Add Data to Stock Charts

To add data to a stock chart, specify the data model in the chart's `value` attribute and configure a `stockDataItem` for each unique group. For details about data requirements, see [Chart Component Data Requirements](#).

The process to add data to funnel charts is similar to the process for adding data to area, bar, horizontal bar, bubble, combination, line, and scatter charts. The primary difference is that you add a `stockDataItem` instead of a `chartDataItem` to the chart.

The example below shows a simple example for configuring a stock chart from a managed bean. In this example, the `getDefaultStockData()` method and the overloaded `chartDataItem` constructor are added to the `chartDataSource` bean referenced in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

```
public ChartDataItem(Object group, Number open, Number close, Number high,
    Number low, Number volume) {
    put("series", "STOCK");
    put("group", group);
}
```

```
        put("open", open);
        put("close", close);
        put("high", high);
        put("low", low);
        put("volume", volume);
    }

    public CollectionModel getDefaultStockData() {
        List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();

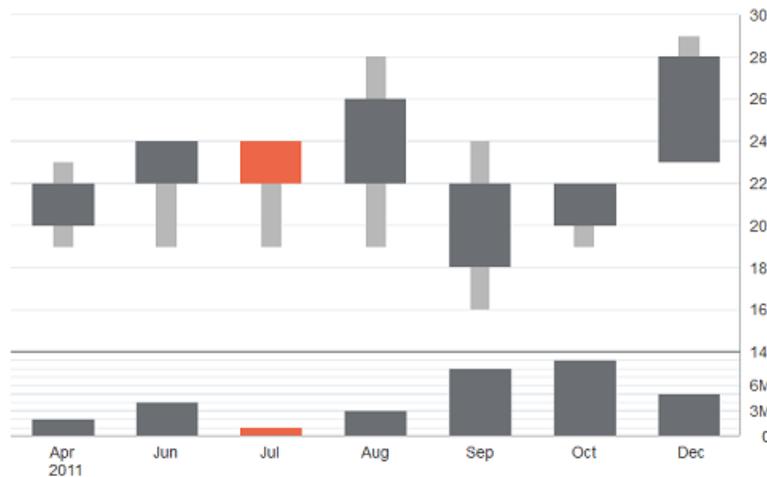
        dataItems.add(new ChartDataItem("Apr 2011",
20.00,22.00,23.00,19.00,2000000));
        dataItems.add(new ChartDataItem("Jun 2011",
22.00,24.00,24.00,19.00,4000000));
        dataItems.add(new ChartDataItem("Jul 2011",
24.00,22.00,24.00,19.00,1000000));
        dataItems.add(new ChartDataItem("Aug 2011",
22.00,26.00,28.00,19.00,3000000));
        dataItems.add(new ChartDataItem("Sep 2011",
18.00,22.00,24.00,16.00,8000000));
        dataItems.add(new ChartDataItem("Oct 2011",
20.00,22.00,22.00,19.00,9000000));
        dataItems.add(new ChartDataItem("Dec 2011",
23.00,28.00,29.00,23.00,5000000));

        return ModelUtils.toCollectionModel(dataItems);
    }
}
```

The example below shows the code on the JSF page that defines a stock chart and references the `ChartDataSource` class. In this example, the managed bean is named `chartDataSource`.

```
<dvt:stockChart value="#{chartDataSource.defaultStockData}" var="row">
    <dvt:stockDataItem series="#{row.series}" group="#{row.group}" volume
    =#{row.volume}" high="#{row.high}"
        low = "#{row.low}" close="#{row.close}"
    open="#{row.open}"/>
</dvt:stockChart>
```

The figure shows the stock chart at run time. In this example, the stock chart has a volume section, and the values are reflected in the y-axis.

Figure 23-44 Stock Chart Configured From Managed Bean

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

Add a stock chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to a stock chart:

- Optionally, create the managed bean that will return the chart's data model.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
- In the Structure window, right-click `dvt:stockChart` and choose **Go To Properties**.
- In the Properties window, do one of the following:
 - To reference a managed bean that returns the chart's data model, expand the **Common** section and specify the EL expression that references the data model in the **Value** field. You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `chartDataSource` that includes the stock data method, as referenced in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#), enter the following in the **Value** field:

```
#{chartDataSource.defaultStockData}.
```

For help with creating EL expressions, see [How to Create an EL Expression](#).

- To bind the chart to a data control, click **Bind to ADF Control** to select a data collection.

For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

4. If you specified a managed bean for the chart's data model, do the following to add a `stockDataItem` to the stock chart:
 - a. Right-click `dvt:stockChart` and choose **Insert Inside Stock > Stock Data Item**.
 - b. In the Structure window, right-click `dvt:stockChart` and choose **Go to Properties**.
 - c. In the Properties window, expand the **Data** section, and enter a value for the **Var** field.

The `var` property is the name of an EL variable that references each element in the data collection. In the second code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#), the `var` attribute is defined as `row`. The relevant code is shown below.

```
<dvt:lineChart id="chart1"
value="#{chartDataSource.weeklyStockData}"
timeAxisType="enabled" var="row">
```

- d. To customize the tooltip that is displayed when the user moves the focus to a data point, enter a value in the **shortDesc** field.
- e. In the Structure window, right-click `dvt:stockDataItem` and choose **Go to Properties**.
- f. In the Properties window, expand the **Data** section. Depending on the type of stock chart, enter values or use the **Expression Builder** from the attribute's drop down menu for the various fields.

For example, to use the `chartDataSource` managed bean, specify these values for the following attributes:

- **Close:** `#{row.close}`
 - **Group:** `#{row.group}`
 - **High:** `#{row.high}`
 - **Low:** `#{row.low}`
 - **Open:** `#{row.open}`
 - **Series:** `#{row.series}`
 - **Volume:** `#{row.volume}`
- g. Configure any additional properties as needed to customize the data item.
 - h. To add additional stock data items, repeat the above steps for each additional data item.

How to Add Data to Spark Charts

Spark charts are simple, charts that display trends or variations, often in the column of a table, or inline with text. Line, bar, and area spark charts require a single series of data values. [Figure 23-45](#) shows an example of a line spark chart in a table column.

Figure 23-45 Line Spark Chart in Table of Stock Prices

Floating bar spark charts require two series of data values, one for the float offset, and one for the bar value. [Figure 23-46](#) shows an example of a floating bar spark chart.

Figure 23-46 Floating Bar Spark Chart

In a simple UI-first development scenario you can insert a spark chart using the Components window and bind it to data afterwards.

You can provide data to spark charts in any of the following ways:

- Specify data statically in child `dvt:sparkItem` tags. The example below shows an example of providing static data to a spark chart.

```
<dvt:sparkChart>
  <dvt:sparkItem value="20" />
  <dvt:sparkItem value="15" />
  <dvt:sparkItem value="30" />
</dvt:sparkChart>
```

- Specify data using EL Expression in child `dvt:sparkItem` tags.

The example below shows an example of providing data to spark charts using EL Expressions to produce the table shown in [Figure 23-45](#).

```
<af:table summary="Table" value="#{sparkChart.tableData}" var="row"
  columnStretching="last"
  rowBandingInterval="1" id="t1" width="230"
  disableColumnReordering="true"
  contentDelivery="immediate" autoHeightRows="8">
  <af:column sortable="false" headerText="Stock Symbol" align="start" id="c1"
    rowHeader="true">
    <af:outputText value="#{row[0]}" id="ot4" />
  </af:column>
  <af:column sortable="false" headerText="Prices for 2014" id="c2">
    <dvt:sparkChart id="sparkchart1" subType="line" axisScaledFromBaseline="off"
      shortDesc="Sparkchart in Table">
      <dvt:sparkItem value="#{row[1]}" id="si6" />
      <dvt:sparkItem value="#{row[2]}" id="si5" />
      <dvt:sparkItem value="#{row[3]}" id="si4" />
      <dvt:sparkItem value="#{row[4]}" id="si3" />
      <dvt:sparkItem value="#{row[5]}" id="si2" />
      <dvt:sparkItem value="#{row[6]}" id="si1" />
    </dvt:sparkChart>
  </af:column>
</af:table>
```

```

    </af:column>
</af:table>

```

In this example, the spark chart's data is defined in a managed bean named `sparkChart`. The example below shows the managed bean. The class is defined as `SparkchartSample`, and the `getTableData()` method defines the columns for the table.

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

public class SparkchartSample {
    public List <List <Object>> getTableData() {
        List <List <Object>> list = new ArrayList <List <Object>>();

        // Create a random number generator with a constant seed
        Random rand = new Random(5);

        String stocks[] = {"ORCL", "AAPL", "MSFT", "YHOO", "CSCO", "PALM", "GOOG",
            "SAP", "HPQ", "IBM", "INTC", "RIMM"};
        Arrays.sort(stocks); // sort to look nice
        for(int i=0; i<stocks.length; i++)
        {
            List <Object> row = new ArrayList <Object>();
            row.add(stocks[i]);
            for(int j=0; j<6; j++) {
                // Let the value vary between 40 and 70 (for simplicity)
                row.add((rand.nextDouble()*30) + 40);
            }
            list.add(row);
        }
        return list;
    }
}

```

- Use `af:iterator` to stamp spark items.

The example below shows an example of using `af:iterator` to provide data to a bar spark chart.

```

<dvt:sparkChart subType="bar" id="sc5">
    <af:iterator value="#{sparkChart.collection}" var="row" id="i3">
        <dvt:sparkItem value="#{row.close}" id="si9"/>
    </af:iterator>
</dvt:sparkChart>

```

In this example, the iteration is defined in the `sparkChart` managed bean. The example below shows the `collection()` method that creates the spark items.

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public List <Map <String, Object>> getCollection() {
    List <Map <String, Object>> list = new ArrayList <Map <String, Object>>();

    // Generate some sort of stock data
    double open[] = {5, 6, 4.5, 6.3, 4.1, 7.6, 8.4, 11.5, 10.5, 11.3};
    double close[] = {6.8, 4.2, 6.7, 4.5, 7.1, 8.6, 10.4, 10.0, 12.5, 14.5};
}

```

```

for(int i=0; i<open.length; i++)
{
    Map <String,Object> row = new HashMap <String,Object>();
    row.put("open", open[i]);
    row.put("close", close[i]);
    list.add(row);
}
return list;
}

```

Figure 23-47 shows the runtime view of the spark chart that displays if you create the spark chart defined in the `af:iterator` examples above.

Figure 23-47 Sparkchart Data Items Stamped by `af:iterator`



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

If you plan on displaying your spark chart in a table, create the ADF table. If you need help creating the table, see [How to Display a Table on a Page](#).

Add a spark chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add data to a spark chart:

1. Create the managed bean that will return the chart's data model.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
2. To use an `af:iterator` to stamp the spark chart data items, do the following:
 - a. Right-click `dvt:sparkChart` and choose **Insert Inside Spark Chart > Iterator**.
 - b. In the Structure window, right-click `af:iterator` and choose **Go to Properties**.
 - c. In the Properties window, expand the **Common** section, and enter a value for the **Value** field.
Reference the EL expression that returns the value for the spark chart data item. To reference the method used in the `collection()` method example above, specify the following for the value: `#{sparkChart.collection}`.
 - d. Enter a value for the **Var** field.
The `var` property is the name of an EL variable that references each element in the data collection. In the `collection()` method example above, the `var` attribute is defined as `row`.
 - e. Configure any additional properties as needed to customize the iterator.

For example, you can specify a maximum number of rows. For more information about working with `af:iterator`, click **Component Help**.

3. To add the spark chart data items, do the following:
 - a. Right-click `dvt:sparkChart` or `af:iterator` and choose **Insert Inside (Spark Chart or Iterator > ADF Data Visualizations) > Spark Item**.
 - b. In the Structure window, right-click `dvt:sparkItem` and choose **Go to Properties**.
 - c. In the Properties window, expand the **Common** section, and enter a value for the **Value** field.

You can enter a static numeric value or reference an EL expression that returns the value for the spark chart data item. To reference the method used in the `collection()` method example above, specify the following for the **Value**: `#{row.close}`.

- d. Configure any additional properties as needed to customize the data item.

For example, you must specify a **FloatValue** for floating bar spark charts. For additional information about the available attributes, click **Component Help**.

- e. To add additional spark chart data items, repeat Step a through Step d for each additional data item.

 **Note:**

You can also bind the spark chart to a data control if your application uses the Fusion technology stack. To do so, click **Bind to ADF Control** in the Properties window to select a data collection. For more information about using data controls to supply data to your chart, see "Creating Databound Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Customizing Chart Display Elements

You can customize most aspects of the ADF DVT Chart component's display, including its labels, legend, axes, series, and animation effects. Some chart types also allow the addition of reference objects, stacking behavior, or a dual Y-Axis.

Pie charts also support the ability to aggregate smaller values or explode a pie slice.

How to Configure Chart Labels

You can customize the style and position of chart data labels on all charts other than spark charts. Depending on the chart type, you can also add a title, subtitle, footnote, legend title, or axis title.

**Tip:**

Use data labels sparingly because they cause clutter and decrease readability. Use data labels primarily to highlight outliers or important values and not to show values for all data points.

How to Configure Chart Data Labels

To configure chart data labels, configure the attributes of the `dvt:chartDataItem` or `dvt:pieDataItem` chart child component.

If you want the style and position of all chart or pie data items to be the same, you can configure attributes directly on the chart component.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

Add data to your chart. For additional information, see [Adding Data to Charts](#).

To configure data labels for charts:

1. In the Structure window, right-click `dvt:chartDataItem` or `dvt:pieDataItem` and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section.
3. To customize the data label colors, enter a value for the following:
 - **Color**: Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter `#008000` to render the data label in green.
 - **BorderColor**: For all charts except spark charts, specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter `#FF0000` to render the data label in red.
 - **BorderWidth**: For all charts except spark charts, specify a numeric value or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the numeric width value.
For example, enter `4` to render the data label border width at 4 units.
4. To customize the position and style of the data label on area, bar, bubble, combination, line, and scatter charts:
 - a. In the Properties window, expand the **Data** section.

How to Configure Chart Element Labels

To configure a chart label, add the chart child component as needed, and configure the style and position of the label. Depending upon the chart type, you can specify labels for the chart's title, subtitle, and footnote.

Note:

Sparkcharts are condensed charts that contain minimal formatting and have little support for labels. Use the spark chart's container to provide descriptive text for the spark chart.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add a title, subtitle or footnote to a chart:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section.
3. To add a title to the chart:

- In the Properties window, enter a value for the title in the **Title** field.

You can enter text for the title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the chart's title.

- From the **TitleHAlign** attribute's dropdown list, select the alignment to use for the title.

By default, the title's alignment is set to `start` which aligns the title to the left of the chart container in left-to-right mode. You can set the alignment to `center`, which positions the title in the center of the chart container, or to `end`, which aligns the title to the right end of the chart container in left-to-right mode.

You can also use `plotAreaStart` to align the title to the left of the plot area in left-to-right mode, `plotAreaCenter` to position the title in the center of the plot area, and `plotAreaEnd` to align the title to the right of the plot area in left-to-right mode.

In right-to-left mode, `start` and `plotAreaStart` will align the title to the right of the chart container and plot area, respectively. `end` and `plotAreaEnd` will align the title to the left of the chart container and plot area, respectively.

4. To add a subtitle to the chart, enter a value in the **Subtitle** field.

You can enter text for the subtitle or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the chart's subtitle.

5. To add a footnote to the chart:

- In the Properties window, enter a value for the footnote in the **Footnote** field.
You can enter text for the footnote or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the chart's footnote.
- From the **FootnoteHAlign** attribute's dropdown list, select the alignment to use for the title.

By default, the footnote's alignment is set to `start` which aligns the footnote to the left of the chart container in left-to-right mode. You can set the alignment to `center`, which positions the footnote in the center of the chart container, or to `end`, which aligns the footnote to the right end of the chart container in left-to-right mode.

You can also use `plotAreaStart` to align the footnote to the left of the plot area in left-to right mode, `plotAreaCenter` to position the footnote in the center of the plot area, and `plotAreaEnd` to align the footnote to the right of the plot area in left-to-right mode.

In right-to-left mode, `start` and `plotAreaStart` will align the footnote to the right of the chart container and plot area, respectively. `end` and `plotAreaEnd` will align the footnote to the left of the chart container and plot area, respectively.

How to Configure Chart Axis Labels

To configure a chart axis label, add a child axis component as needed, and configure the style and position of the label. Depending on the chart type, you can specify the legend title, or axis title and tick labels.

 **Note:**

Sparkcharts are condensed charts that contain minimal formatting and have little support for labels. Use the spark chart's container to provide descriptive text for the spark chart.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure chart axis labels:

1. In the structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > (Chart X Axis or Chart Y Axis or Chart Y2 Axis)**.

2. Right-click the axis you added in the previous step and choose **Go to Properties**.
For example, right-click **ChartXAxis** and choose **Go to Properties**.

3. To add a title to the axis, in the Properties window, expand the **Common** section and enter a value in the **Title** field.

You can enter text for the axis title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the axis title.

4. To configure axis tick labels, in the Structure window, right-click the **dvt:chartXAxis**, **dvt:chartYAxis**, or **dvt:ChartY2Axis** node and choose **Insert Inside Axis > Chart Tick Label**.

5. Right-click **dvt:chartTickLabel** and choose **Go to Properties**.

6. In the Properties window, enter values for the following:

- **LabelStyle**: Enter the CSS attribute to use for the axis label.

For example, if you want to specify a bold font for the axis label, enter the following for the **InlineStyle** attribute:

```
font-weight:bold;
```

- **Position**: From the Position attribute's dropdown list, select `inside` to position the labels inside the chart.

By default, the chart will automatically position the labels outside the axis.

- **Rotation**: From the **Rotation** attribute's dropdown list, select `off` to turn off rotation.

By default, the chart will automatically rotate the labels by 90 degrees if needed to fit more labels on the axis. The rotation will only be applied to categorical labels for a horizontal axis.

- **Scaling**: From the attribute's dropdown list, select the scale factor for the numeric value.

Scaling options range from `none` to `quadrillion`.

For information about customizing chart legend titles, see [How to Configure Chart Legends](#).

How to Configure Chart Legends

You can add a legend to all charts except the spark chart to identify the chart's series and associated colors. If you chose a quick start layout that included a legend when you created your chart, you should already have a legend on the page.

After you have added the legend, you can customize its position, configure a legend title, or disable scrolling.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure a chart legend:

1. In the Structure window, expand the **dvt:typeChart** node.
2. If the **dvt:chartLegend** node is missing in the expanded display, right-click **dvt:typeChart** and choose **Insert Inside Chart > Chart Legend**.
3. Right-click **dvt:chartLegend** and choose **Go to Properties**.
4. In the Properties window, enter values in the following fields as needed.
 - **Position:** From the **Position** attribute's dropdown list, select a position for the legend.

By default, the legend's position is set to `auto` which will position the legend on the side or bottom of the chart, depending upon the size and content of the chart.

You can set the position to `start`, which aligns the legend to the left in left-to-right mode and aligns the legend to the right in right-to-left mode, or `end` which aligns the legend to the right in left-to-right mode and aligns the legend to the left in right-to-left mode.

You can also set the alignment to `top` which positions the legend at the top of the chart or `bottom` to position the legend at the bottom of the chart.

- **SymbolWidth:** Enter a common width in pixels for all symbols in the legend.
You can enter text for the legend title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the legend title.
- **SymbolHeight:** Enter a common height in pixels for all symbols in the legend.
You can enter text for the legend title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the legend title.
- **Title:** Enter a title for the legend.
You can enter text for the legend title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the legend title.
- **TitleHAlign:** From the **TitleHAlign** attribute's dropdown list, select an alignment for the legend's title.

By default, the legend's alignment is set to `start` which aligns the legend's title to the left in left-to-right mode and aligns the legend to the right in right-to-left mode.

You can set the alignment to `center` which positions the legend's title in the center. You can also set the alignment to `end` which aligns the legend's title to the right in left-to-right mode and aligns the title to the left in right-to-left mode.

- **ReferenceObjectTitle:** Enter a title for the reference object area, if one exists.

You can enter text for the title or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the title.

For information about adding reference objects to your chart, see [Adding Reference Objects to a Chart](#).

- **Scrolling:** From the **Scrolling** attribute's dropdown list, select the legend's scrolling behavior.

By default, the chart's scrolling behavior is set to `asNeeded` which will add a scrollbar to the legend if needed. You can select `off` to disable legend scrolling.

- **Size** and **MaximumSize:** Enter the size and maximum size of the axis in pixels or percentage. By default, the size and maximum size values are not provided.

How to Format Chart Numerical Values

You can customize the appearance of numeric values on charts using the `dvt:chartValueFormat` or `dvt:chartTickLabel` tags.

[Figure 23-48](#) shows the effect of adding `dvt:chartValueFormat` and `dvt:chartTickLabel` tags to a bar chart configured to display monthly salaries for a fictitious department. In this example, the top bar chart is using default numeric formatting. The bar chart at the bottom of the figure is configured to display the series value and axis label as currency values.

Figure 23-48 Bar Chart Configured to Display Currency Values



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

How to Format Numeric Values on a Chart's Value, Value Label, or Axis Values

Use the `dvt:chartValueFormat` tag to format numeric values for the chart's value, value label, or axis values.

The `dvt:chartValueFormat` tag allows you to specify the scaling for numeric display.

To format numeric values on a chart's value, value label, or axis values:

1. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Chart Value Format**.
2. Right-click the **dvt:chartValueFormat** node and choose **Go to Properties**.
3. In the Properties window, enter values for the following:
 - **Type:** From the attribute's dropdown list, select the tag that identifies the chart element that you wish to format. Valid values include `x`, `y`, `y2`, `z`, `value`, or `label`.

For example, to format the numeric value for the bar chart's series shown in [Figure 23-48](#), select `y` from the **Type** attribute's dropdown list.
 - **Scaling:** From the attribute's dropdown list, select the scale factor for the numeric value.

Scaling options range from `none` to `quadrillion`.
 - **TooltipDisplay:** From the attribute's dropdown list, select whether a custom tooltip label should be displayed or not. Valid values are `off` or `auto`. The behavior of `auto` differs across chart types.
 - **TooltipLabel:** Specify a text value or use the **Expression Builder** to set custom labels for tooltip values.

How to Format Numeric Values on a Chart's Axis Label

Use the `dvt:chartTickLabel` tag to format the values on the axis labels.

The `dvt:chartTickLabel` tag allows you to specify the scaling for numeric display. It also has a rotation property to specify whether the chart will automatically rotate the labels by 90 degrees in order to fit more labels on the axis. The rotation will only be applied to categorical labels for a horizontal axis. It also has a position property to specify whether the axis labels will be positioned inside or outside the chart.

To format numeric values on a chart's axis label:

1. In the Structure window, expand the **dvt:typeChart** node.
2. If the expanded node does not contain the axis that you wish to format, right-click **dvt:typeChart** and choose **Insert Inside Chart > (Chart X Axis or Chart Y Axis or Chart Y2 Axis)**.
3. Right-click the **dvt:chartXAxis**, **dvt:chartYAxis**, or **dvt:ChartY2Axis** node and choose **Insert Inside Axis > Chart Tick Label**.
4. Right-click **dvt:chartTickLabel** and choose **Go to Properties**.
5. In the Properties window, enter values for the following:
 - **Rotation:** From the **Rotation** attribute's dropdown list, select `off` to turn off rotation.

By default, the chart will automatically rotate the labels by 90 degrees in order to fit more labels on the axis. The rotation will only be applied to categorical labels for a horizontal axis.

- **Scaling:** From the attribute's dropdown list, select the scale factor for the numeric value.

Scaling options range from `none` to `quadrillion`.

How to Specify Numeric Patterns, Currency, or Percent

You can specify specific patterns for the numeric values or change the numeric display to currency or percent by adding the `af:convertNumber` tag to the `dvt:chartValueFormat` or `dvt:chartTickLabel` tags. For additional information about using the `af:convertNumber` tag, see [Validating and Converting Input](#).

To specify numeric patterns, currency, or percent using `af:convertNumber`:

1. In the Structure window, right-click `dvt:chartTickLabel` or `dvt:chartValueFormat` and choose **Insert Inside (Chart Tick Label or Chart Value Format) > Convert Number**.
2. Right-click `af:convertNumber` and choose **Go to Properties**.
3. In the Properties window, from the **Type** attribute's dropdown list, select the desired numeric type.

By default, **Type** is set to `number`, but you can also select `currency` or `percent`.

4. Configure additional attributes as needed.

For example, you can specify which currency symbol to use in the **CurrencySymbol** field. For help with an individual field or to look at the complete tag documentation for the `af:convertNumber` tag, click **Component Help**.

Customizing a Chart Axis

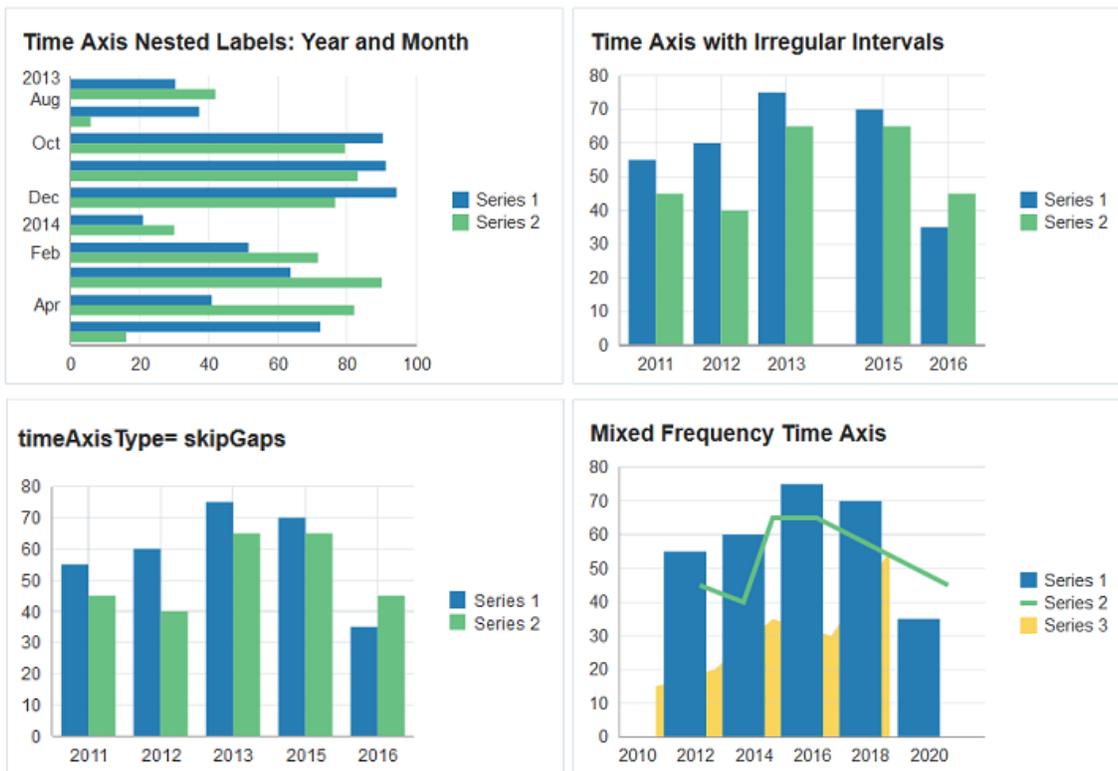
Depending upon the chart type, you can configure a time axis on the x-axis or customize the properties of a chart's x-axis, y-axis, y2-axis, or z-axis.

How to Configure a Time Axis

If your x-axis includes time data, then you can enable the time axis display. Charts include support for time data that is based on regular or irregular intervals. Charts also support mixed frequency time data, where the individual series contain differing dates.

[Figure 23-49](#) shows four charts configured for a time axis. In the horizontal bar chart, the time data is spaced equally. The vertical bar chart shows an example of irregularly spaced data, and the chart provides a visual clue that there is no data for 2014. The second vertical bar chart does not show the visual clue for the missing data. The combination chart shows mixed frequency time data. In this example, the time data is different for each series (chart) in the combination chart.

Figure 23-49 Charts Configured for Regular, Irregular, and Mixed Frequency Time Data



To configure a time axis, set the `timeAxisType` attribute's value to `enabled` for regular or irregular data, set it to `skipGaps` to not indicate empty space in irregular data, and set it to `mixedFrequency` for time data that varies by series.

The example below shows the code snippet on the JSF page that defines the horizontal bar chart, bar charts, and combination chart, with the `timeAxisType` definition highlighted in bold font.

```
<af:panelBox text="Time Axis Nested Labels: Year and Month"
  showDisclosure="false">
  <dvt:barChart orientation="horizontal" value="#{chartDataSource.monthlyTimeData}"
    var="row" hoverBehavior="dim" timeAxisType="enabled">
    <f:facet name="dataStamp">
      <dvt:chartDataItem series="#{row.series}" group="#{row.group}"
        value="#{row.value}"/>
    </f:facet>
  </dvt:barChart>
</af:panelBox>
<af:panelBox text="Time Axis with Irregular Intervals" showDisclosure="false">
  <dvt:barChart value="#{chartDataSource.yearlyIrregularTimeData}"
    var="row" hoverBehavior="dim" timeAxisType="enabled">
    <f:facet name="dataStamp">
      <dvt:chartDataItem series="#{row.series}" group="#{row.group}"
        value="#{row.value}"/>
    </f:facet>
  <dvt:chartXAxis>
    <dvt:majorTick rendered="true"/>
  </dvt:chartXAxis>
```

```

        </dvt:barChart>
    </af:panelBox>
    <af:panelBox text="timeAxisType= skipGaps" showDisclosure="false">
        <dvt:barChart value="#{chartDataSource.yearlyIrregularTimeData}" var="row"
        hoverBehavior="dim" timeAxisType="skipGaps">
            <f:facet name="dataStamp">
                <dvt:chartDataItem series="#{row.series}" group="#{row.group}"
                value="#{row.value}"/>
            </f:facet>
            <dvt:chartXAxis>
                <dvt:majorTick rendered="true"/>
            </dvt:chartXAxis>
        </dvt:barChart>
    </af:panelBox>
    <af:panelBox text="Mixed Frequency Time Axis" showDisclosure="false">
        <dvt:comboChart value="#{chartDataSource.yearlyMixedFrequencyTimeData}"
        var="row" hoverBehavior="dim" timeAxisType="mixedFrequency">
            <f:facet name="dataStamp">
                <dvt:chartDataItem series="#{row.series}" group="#{row.group}" x="#{row.x}"
                y="#{row.y}"/>
            </f:facet>
        </dvt:comboChart>
    </af:panelBox>

```

The code that defines the charts' collection models and populate them with data is stored in the `chartDataSource` managed bean shown in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

The example below shows the `getMonthlyTimeData()`, `getYearlyIrregularTimeData()`, and `getYearlyMixedFrequencyTimeData()` methods. In this example, the `getMonthlyTimeData()` method calls the `getTimeData()` method which is a reusable method that takes the series, group, and time data as its arguments.

```

public CollectionModel getYearlyIrregularTimeData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    dataItems.add(new ChartDataItem("Series 1", new GregorianCalendar(2011, 7,
27).getTime(), 55));
    dataItems.add(new ChartDataItem("Series 1", new GregorianCalendar(2012, 7,
27).getTime(), 60));
    dataItems.add(new ChartDataItem("Series 1", new GregorianCalendar(2013, 7,
27).getTime(), 75));
    dataItems.add(new ChartDataItem("Series 1", new GregorianCalendar(2015, 1,
27).getTime(), 70));
    dataItems.add(new ChartDataItem("Series 1", new GregorianCalendar(2016, 1,
27).getTime(), 35));

    dataItems.add(new ChartDataItem("Series 2", new GregorianCalendar(2011, 7,
27).getTime(), 45));
    dataItems.add(new ChartDataItem("Series 2", new GregorianCalendar(2012, 7,
27).getTime(), 40));
    dataItems.add(new ChartDataItem("Series 2", new GregorianCalendar(2013, 7,
27).getTime(), 65));
    dataItems.add(new ChartDataItem("Series 2", new GregorianCalendar(2015, 1,
27).getTime(), 65));
    dataItems.add(new ChartDataItem("Series 2", new GregorianCalendar(2016, 1,
27).getTime(), 45));

    return ModelUtils.toCollectionModel(dataItems);
}
public CollectionModel getYearlyMixedFrequencyTimeData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();

```

```

        dataItems.add(new ChartDataItem("Series 1", "Group 1", new GregorianCalendar(2011,
7, 27).getTime(), 55));
        dataItems.add(new ChartDataItem("Series 1", "Group 2", new GregorianCalendar(2013,
7, 27).getTime(), 60));
        dataItems.add(new ChartDataItem("Series 1", "Group 3", new GregorianCalendar(2015,
7, 27).getTime(), 75));
        dataItems.add(new ChartDataItem("Series 1", "Group 4", new GregorianCalendar(2017,
7, 27).getTime(), 70));
        dataItems.add(new ChartDataItem("Series 1", "Group 5", new GregorianCalendar(2019,
7, 27).getTime(), 35));

        dataItems.add(new ChartDataItem("Series 2", "Group 1", new GregorianCalendar(2012,
1, 27).getTime(), 45));
        dataItems.add(new ChartDataItem("Series 2", "Group 2", new GregorianCalendar(2013,
7, 27).getTime(), 40));
        dataItems.add(new ChartDataItem("Series 2", "Group 3", new GregorianCalendar(2014,
7, 27).getTime(), 65));
        dataItems.add(new ChartDataItem("Series 2", "Group 4", new GregorianCalendar(2016,
1, 27).getTime(), 65));
        dataItems.add(new ChartDataItem("Series 2", "Group 5", new GregorianCalendar(2020,
7, 27).getTime(), 45));
        dataItems.add(new ChartDataItem("Series 3", "Group 1", new GregorianCalendar(2010,
7, 27).getTime(), 15));
        dataItems.add(new ChartDataItem("Series 3", "Group 2", new GregorianCalendar(2012,
7, 27).getTime(), 20));
        dataItems.add(new ChartDataItem("Series 3", "Group 3", new GregorianCalendar(2014,
7, 27).getTime(), 35));
        dataItems.add(new ChartDataItem("Series 3", "Group 4", new GregorianCalendar(2016,
7, 27).getTime(), 30));
        dataItems.add(new ChartDataItem("Series 3", "Group 5", new GregorianCalendar(2018,
7, 27).getTime(), 55));
        return ModelUtils.toCollectionModel(dataItems);
    }
    public CollectionModel getMonthlyTimeData() {
        return getTimeData(2, 10, 2013, 7, 1, Calendar.MONTH, 1);
    }
    private double getValue() {return Math.random() * 100;}
    public CollectionModel getTimeData(int numSeries, int numGroups,
        int startYear, int startMonth, int startDate,
        int dateField, int addCount){
        List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
        GregorianCalendar cal = new GregorianCalendar(startYear, startMonth, startDate);
        for(int group=0; group<numGroups; group++) {
            for(int series=0; series<numSeries; series++) {
                dataItems.add(new ChartDataItem("Series " + (series+1), cal.getTime(),
getValue()));
            }
            cal.add(dateField, addCount);
        }
        return ModelUtils.toCollectionModel(dataItems);
    }
}

```

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure a chart time axis:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
2. In the Properties window, expand the **Appearance** section.
3. From the **TimeAxisType** attribute's dropdown list, select `enabled` if your chart includes regular or irregular time data. Select `skipGaps` if your chart includes irregular time data where the time skip should not be highlighted, such as in stock data where the weekend periods do not matter. Select `mixedFrequency` if your chart contains mixed frequency time data.

How to Customize the Chart Axis

Charts include support for customizing the x-axis, y-axis, and y2-axis. Depending upon the axis type, you can customize the minimum and maximum values of the data and axis, add a title, specify tick mark increments, configure viewport boundaries, and specify whether the axis baseline starts at the minimum value of the data or at zero.

To customize a chart axis, add the `dvt:chartXAxis`, `dvt:chartYAxis`, or `dvt:chartY2Axis` component to the chart and configure the properties for the axis in the Properties window.

[Table 23-6](#) lists the chart axis attributes and the axis on which they apply. Most attributes have default settings which are based upon the data, and you only need to change these if you want to modify the defaults. However, you must specify values for the `title`, `attributeChangeListener`, and `binding` attributes if you want to use them.

Table 23-6 Chart Axis Attributes

Name	Description	X-Axis Support?	Y-Axis Support?	Y2-Axis Support?
<code>alignTickMarks</code>	Specifies whether the tick marks of the y1 and y2 axes are aligned.	No	No	Yes
<code>attributeChangeListener</code>	Method reference to a listener for renderer changes to a property without the application's specific request.	Yes	Yes	Yes
<code>baselineScaling</code>	Specifies whether the axis baseline starts at the minimum value of the data or at zero.	Yes	Yes	Yes
<code>binding</code>	Specifies a binding reference to store a specific instance of the axis from a backing bean. Set this attribute only to access code in a backing bean.	Yes	Yes	Yes
<code>dataMaximum</code>	Specifies the maximum data value on a numerical axis. If not set, attribute defaults to the maximum value of the data set.	Yes	Yes	Yes
<code>dataMinimum</code>	Specifies the minimum data value on a numerical axis. If not set, attribute defaults to the maximum value of the data set.	Yes	Yes	Yes

Table 23-6 (Cont.) Chart Axis Attributes

Name	Description	X-Axis Support?	Y-Axis Support?	Y2-Axis Support?
id	Specifies the component's identifier.	Yes	Yes	Yes
majorIncrement	Specifies the increment for the major ticks of a numerical axis.	Yes	Yes	Yes
maximum	Specifies the maximum value of the axis.	Yes	Yes	Yes
maximumSize	Specifies the maximum width or height of the axis	Yes	Yes	Yes
minimum	Specifies the minimum value of the axis.	Yes	Yes	Yes
minimumIncrement	Specifies the minimum increment between tick marks on a numerical axis.	Yes	Yes	Yes
minorIncrement	Specifies the increment for the minor ticks of the axis.	Yes	Yes	Yes
position	Specifies the location of the axis label in relation to the plot area.	No	Yes	Yes
rendered	Specifies whether the axis is rendered.	Yes	Yes	Yes
scale	Specifies whether the axis is linear or logarithmic	Yes	Yes	Yes
size	Specifies the width or height of the axis	Yes	Yes	Yes
title	Specifies the title of the axis.	Yes	Yes	Yes
viewportEndGroup	Specifies the end group of the current viewport on group or time axes.	Yes	No	No
viewportStartGroup	Specifies the start group of the current viewport on group or time axes.	Yes	No	No
viewportMaximum	Specifies the maximum x-coordinate or y-coordinate of the current viewport on bubble and scatter charts.	Yes	Yes	No
viewportMinimum	Specifies the minimum x-coordinate of the current viewport	Yes	Yes	No

Chart axes also support optional child components which allow you to customize the major and minor tick marks, axis tick labels, and axis lines.

- `dvt:majorTick` and `dvt:minorTick`: Specifies the line color, style, and width and baseline color, style and width of the chart's tick marks.
- `dvt:chartAxisLine`: Specifies the line color and width of the axis line.
- `dvt:chartTickLabel`: Specifies the scaling, styling, and rotation of the chart's tick labels.

For details about configuring `dvt:chartTickLabel`, see [How to Format Chart Numerical Values](#) and [How to Configure Chart Element Labels](#).

- `dvt:referenceArea` and `dvt:referenceLine`: Specifies a reference area or line on the axis.

For information about adding a reference area or line to your chart, see [Adding Reference Objects to a Chart](#).

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To customize a chart axis, axis line, and axis major or minor ticks:

1. In the Structure window, expand the **dvt:typeChart** node.
2. If the expanded node does not contain the axis that you wish to customize, right-click **dvt:typeChart** and choose **Insert Inside Chart > (Chart X Axis or Chart Y Axis or Chart Y2 Axis)**.
3. Right-click the **dvt:chartXAxis**, **dvt:chartYAxis**, or **dvt:ChartY2Axis** node and choose **Go to Properties**.
4. To customize a numerical axis, in the Properties window, enter values for the following:
 - **BaselineScaling**: From the **BaselineScaling's** attribute's dropdown list, select `min` to set the axis baseline to the minimum value of the data. By default, this attribute is set to `zero`.
 - **DataMaximum** and **DataMinimum**: Enter the minimum and maximum values for which data will be displayed.
By default, the chart will display the entire data set.
 - **MajorTickIncrement** and **MinorTickIncrement**: Specify the increments for the major and minor ticks.
By default, the tick increments are calculated from the data.
 - **Scale**: From the attribute's dropdown list, select the scale type for the numeric value.
You can select either `linear` or `log`.
5. For all axis types, in the Properties window, enter values for the following attributes.
 - **Minimum** and **Maximum**: Enter the minimum and maximum axis values.
By default, the minimum and maximum values are calculated from the data.
 - **Size** and **MaximumSize**: Enter the size and maximum size of the axis in pixels or percentage. By default, the size and maximum size values are not provided.
 - **Title**: Specify the axis title. By default, the axis does not include a title.
6. To configure a viewport, in the Properties window, enter values for the **viewportEndGroup**, **viewportStartGroup**, **viewportMaximum**, and **viewportMinimum** attributes as needed.

For information about adding a viewport to your chart, see [How to Configure Chart Zoom and Scroll](#).

7. Configure additional attributes as needed.

For example, you can turn off alignment of the tick marks on the y1 and y2 axes of a dual-Y chart using the **AlignTickMarks** attribute. For help with an individual field or to look at the complete tag documentation for the axis, click **Component Help**.

8. Right-click the **dvt:chartXAxis**, **dvt:chartYAxis**, or **dvt:ChartY2Axis** node and choose **Insert Inside Axis > Chart Tick Label**.
9. Right-click **dvt:chartTickLabel** and choose **Go to Properties**.
10. In the Properties window, enter values for the following attributes.
 - **Position:** From the Position attribute's dropdown list, select `inside` to position the labels inside the chart.
By default, the chart will automatically position the labels outside the axis.
 - **Rotation:** From the **Rotation** attribute's dropdown list, select `off` to turn off rotation.
By default, the chart will automatically rotate the labels by 90 degrees in order to fit more labels on the axis. The rotation will only be applied to categorical labels for a horizontal axis.
 - **Scaling:** From the attribute's dropdown list, select the scale factor for the numeric value.
Scaling options range from `none` to `quadrillion`.

11. To further customize the numeric value, in the Structure window, right-click **dvt:chartTickLabel** and choose **Insert Inside Chart Tick Label > Convert Number**.

12. Right-click **af:convertNumber** and choose **Go to Properties**.

13. In the Properties window, from the **Type** attribute's dropdown list, select the desired numeric type.

By default, **Type** is set to `number`, but you can also select `currency` or `percent`.

14. Configure additional attributes as needed.

For example, you can specify which currency symbol to use in the **CurrencySymbol** field. For help with an individual field or to look at the complete tag documentation for the **af:convertNumber** tag, click **Component Help**.

15. To customize the chart's axis line, in the Structure window, right-click the **dvt:chartXAxis**, **dvt:chartYAxis**, or **dvt:ChartY2Axis** node and choose **Insert Inside Axis > Chart Axis Line**.

16. Right-click **dvt:chartAxisLine** and choose **Go to Properties**.

17. In the Properties window, enter values for the following attributes.

- **LineColor:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter `#008000` to render the axis line's color in green.
- **LineWidth:** Enter the axis line's width in pixels. By default, **LineWidth** is set to 1.

18. To customize the axis major or minor ticks, in the Structure window, right-click the `dvt:chartXAxis`, `dvt:chartYAxis`, or `dvt:ChartY2Axis` node and choose **Insert Inside Axis > (Major Tick or Minor Tick)**.
19. Right-click `dvt:majorTick` or `dvt:minorTick` and choose **Go to Properties**.
20. In the Properties window, enter values for the following attributes.
 - **BaselineColor**: Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter `#800000` to render the baseline color in red.
 - **BaselineStyle**: From the `BaselineStyle` attribute's dropdown list, select the tick mark's baseline style.
BaselineStyle accepts the values `dashed`, `dotted` or `solid`.
 - **BaselineWidth**: Enter the tick mark's baseline width in pixels.
 - **LineColor**: Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter `#008000` to render the tick mark's color in green.
 - **LineStyle**: From the `LineStyle` attribute's dropdown list, select the tick mark's line style.
By default, **LineStyle** is set to `solid`, but you can select `dashed` or `dotted` to change the default line style.
 - **LineWidth**: Enter the tick mark's width in pixels. By default, **LineWidth** is set to 1.

Configuring Dual Y-Axis

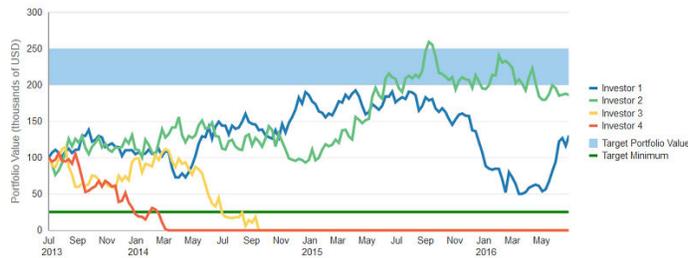
You can add a second y-axis to area, bar, combination, and line charts. This feature is useful if you have two series with different ranges of data. For example, you could have a bar chart that shows salary on one y-axis and commission percent on the second.

To configure a dual y-axis, add the `dvt:chartSeriesStyle` component to your chart, and set the `assignedToY2` attribute to `true`. For instructions to configure the `dvt:chartSeriesStyle` component, see [How to Customize a Chart Series](#).

Adding Reference Objects to a Chart

You can add reference areas or lines to all charts except pie charts using the `dvt:referenceLine` or `dvt:referenceArea` components. Reference areas are associated with the chart's axis and typically are associated with the y-axis. Use reference areas or lines to show target values or ranges.

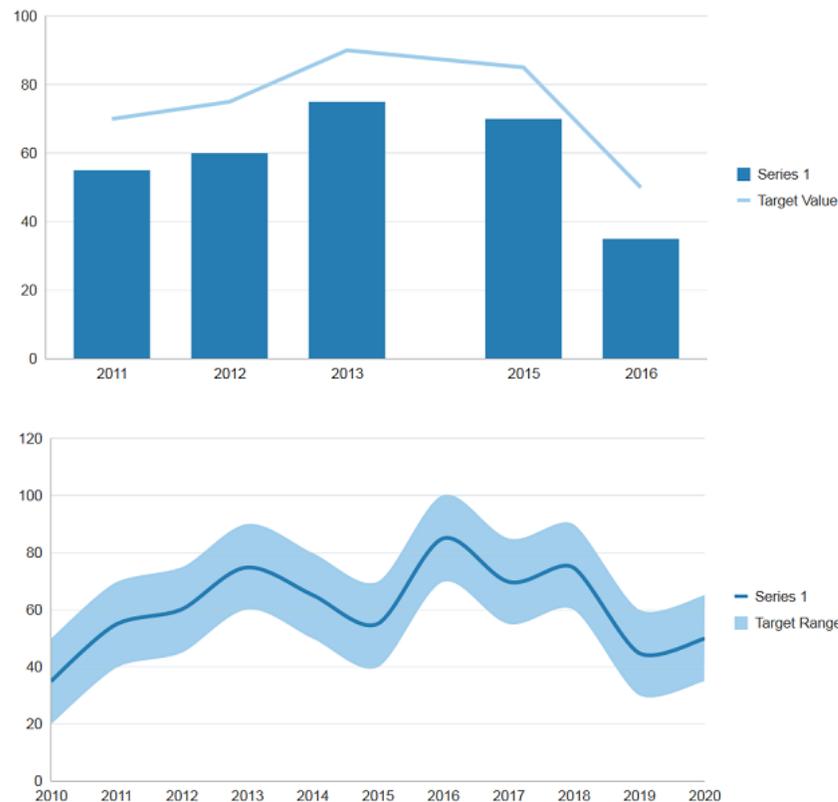
[Figure 23-50](#) shows the line chart displayed in [Figure 23-40](#) with a reference area and reference line. In this example, the reference area is configured with a light blue color and a range between 200 and 250. The reference line is configured with a green color and a value of 25.

Figure 23-50 Line Chart Configured With Reference Area and Reference Line

The example below shows the code snippet on the JSF page that defines the reference area and line. In this example, the reference area and line are configured as children of the chart's y-axis.

```
<dvt:chartYAxis title="Portfolio Value (thousands of USD)">
  <dvt:referenceArea id="ral" color="#A0CEEC" maximum="250" minimum="200"
    displayInLegend="on" text="Target Portfolio Value"/>
  <dvt:referenceLine id="rll" color="#008000" value="25" displayInLegend="on"
    text="Target Minimum" rendered="true" lineWidth="3"/>
</dvt:chartYAxis>
```

You can also specify multi-segment reference lines or areas as shown in [Figure 23-51](#). In this example, the bar chart is configured with a multi-segment reference line, and the line chart is configured with a multi-segment reference area. In these examples, the target values vary by year.

Figure 23-51 Multi-Segment Reference Line and Reference Area

Multi-segment reference lines or reference areas are defined by adding the `dvt:referenceAreaItem` or `dvt:referenceLineItem` elements as children of the `dvt:referenceArea` and `dvt:referenceLine` components. You can explicitly define a reference area or line item for each segment in the reference line or area, or use the `af:iterator` tag to loop through the segments.

The example below shows the code snippets on the JSF page that define the bar and line charts. The code related to the reference area and reference line is highlighted in bold.

```
<dvt:barChart value="#{chartDataSource.yearlyIrregularTimeData}"
    var="row" timeAxisType="enabled" inlineStyle="width:650px">
  <f:facet name="dataStamp">
    <dvt:chartDataItem series="Series 1" group="#{row.group}"
      value="#{row.value}"/>
  </f:facet>
  <dvt:chartYAxis maximum="100">
    <b>dvt:referenceLine color="#A0CEEC" displayInLegend="on" text="Target
    Value"
      lineWidth="3" shortDesc="Target Value">
      <b><af:iterator id="it1"
value="#{chartDataSource.yearlyIrregularTimeData}"
      var="row">
        <dvt:referenceLineItem x="#{row.series == 'Series 1' ? row.group:
0}"
          value="#{row.series == 'Series 1' ? row.value +
```

```

15:0}"/>
    </af:iterator>
    </dvt:referenceLine>
    </dvt:chartYAxis>
    <dvt:chartLegend rendered="true"/>
</dvt:barChart>
    ...
<dvt:lineChart value="#{chartDataSource.yearlySingleTimeData}" var="row"
    timeAxisType="enabled" inlineStyle="width:650px" id="chart1">
    <f:facet name="dataStamp">
        <dvt:chartDataItem series="#{row.series}" group="#{row.group}"
            value="#{row.value}"/>
    </f:facet>
    <dvt:chartYAxis>
        <dvt:referenceArea color="#A0CEEC" displayInLegend="on" text="Target
Range"
            shortDesc="Target Range">
            <af:iterator id="it3" value="#{chartDataSource.yearlySingleTimeData}"
                var="row">
                <dvt:referenceAreaItem x="#{row.group}" minimum="#{row.value - 15}"
                    maximum="#{row.value + 15}"/>
            </af:iterator>
        </dvt:referenceArea>
    </dvt:chartYAxis>
    <dvt:chartLegend rendered="true"/>
</dvt:lineChart>

```

The data for the bar chart is defined in the `getYearlyIrregularTimeData()` method and is shown in the second code sample in [Customizing a Chart Axis](#), and the line chart's data is defined in the `getYearlySingleTimeData()` method. Both methods are contained in the `chartDataSource` managed bean.

The code below shows the `getYearlySingleTimeData()` method that defines the line chart's collection model and populates it with sample data.

```

public CollectionModel getYearlySingleTimeData() {
    List<ChartDataItem> dataItems = new ArrayList<ChartDataItem>();
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2010, 1, 27).getTime(), 35));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2011, 1, 27).getTime(), 55));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2012, 1, 27).getTime(), 60));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2013, 1, 27).getTime(), 75));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2014, 1, 27).getTime(), 65));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2015, 1, 27).getTime(), 55));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2016, 1, 27).getTime(), 85));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2017, 1, 27).getTime(), 70));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2018, 1, 27).getTime(), 75));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2019, 1, 27).getTime(), 45));
    dataItems.add(new ChartDataItem("Series 1",
        new GregorianCalendar(2020, 1, 27).getTime(), 50));
}

```

```
    return ModelUtils.toCollectionModel(dataItems);  
}
```

How to Add a Reference Object to a Chart

To add a reference object to a chart, add the `dvt:referenceArea` or `dvt:referenceLine` component as a child of the chart's associated axis. The process is the same for all charts except the spark charts which use reference objects wrapped inside a reference object set for reference areas and lines.

To specify segmented reference lines or areas, add the `dvt:referenceAreaItem` or `dvt:referenceLineItem` as a child of the `dvt:referenceArea` or `dvt:referenceLine`. To use `af:iterator` to loop through the reference items, add the `af:iterator` as a child of the chart's axis and then add the reference item.

Note:

To add a reference line or object to a spark chart, add the `dvt:referenceObjectSet` component to the spark chart and configure a `dvt:referenceObject` for each reference line or area. Consult the tag documentation for additional information.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add a reference line or reference area to a chart:

1. In the Structure window, expand the **`dvt:typeChart`** node.
2. If the expanded node does not contain the axis that will contain the reference line, right-click **`dvt:typeChart`** and choose **Insert Inside Chart > (Chart X Axis or Chart Y Axis or Chart Y2 Axis)**.
3. Right click the **`dvt:axis`** node and choose **Insert Inside Axis > Reference Line** for a reference line or choose **Insert Inside Axis > Reference Area** for a reference area.
4. Right-click **`dvt:referenceLine`** or **`dvt:referenceArea`** and choose Go to Properties.
5. In the Properties window of the reference line, enter values for the following:
 - **Value:** Specify the value on the axis where the line is to appear.
For example, enter 25 to position the line where the axis equals 25.
 - **Color:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.

For example, enter #008000 to display the reference line in green.

- **LineStyle:** From the attribute's dropdown list, select dashed or dotted to change the default display from solid.
- **LineType:** From the attribute's dropdown list, select a line type. By default, this attribute is set to auto. You can also set this to straight, curved, stepped, centeredStepped, segmented, centeredSegmented or none.
- **LineWidth:** Enter the width in pixels for the line.
- **Text:** Enter a description for the reference line.
- **DisplayedInLegend:** From the attribute's dropdown list, select on to have the reference line included in the chart's legend.

In the Properties window of the reference area, enter values for the following:

- **Maximum:** Specify the upper bound of the reference area.
For example, enter 300 to set the upper bound of the reference area to 300 along the chart's associated axis.
- **Minimum:** Specify the lower bound of the reference area.
For example, enter 250 to set the lower bound of the reference area to 250 along the chart's associated axis.
- **Color:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter #A0CEEC to display the reference line in light blue.
- **LineType:** From the attribute's dropdown list, select a line type. By default, this attribute is set to auto. You can also set this to straight, curved, stepped, centeredStepped, segmented, centeredSegmented or none.
- **Text:** Enter a description for the reference line.
- **DisplayedInLegend:** From the attribute's dropdown list, select on to have the reference area included in the chart's legend.

6. In the Structure window, to specify a segmented reference line and define the reference line items using `af:iterator`, right-click `dvt:referenceLine` and choose **Insert Inside Reference Line Item > Iterator**.

In the Structure window, to specify a segmented reference area and define the reference area items using `af:iterator`, right-click `dvt:referenceArea` and choose **Insert Inside Reference Area Item > Iterator**.

7. If working with a segmented reference line, right-click the `af:iterator` or `dvt:referenceLine` node and choose **Insert Inside (Iterator or Reference Line) > Reference Line Item**.

If working with a segmented reference area, right-click the `af:iterator` or `dvt:referenceArea` node and choose **Insert Inside (Iterator or Reference Area) > Reference Area Item**.

8. Right-click `dvt:referenceLineItem` or `dvt:referenceAreaItem` and choose **Go to Properties**.

9. In the Properties window, enter values for the following:
 - **Value:** Specify the value of the reference line item.

You can enter a static value or use an EL Expression that evaluates to the reference line item's value. For example, to specify the value for the reference line item for the line chart shown in [Figure 23-51](#), enter `#{row.series == 'Series 1' ? row.value + 15:0}`.

- **Maximum:** Specify the maximum value of the reference area item.

You can enter a static value or use an EL Expression that evaluates to the reference area item's value. For example, to specify the maximum value for the reference area item for the line chart shown in [Figure 23-51](#), enter `#{row.value + 15}`.

- **Minimum:** Specify the minimum value of the reference area item.

You can enter a static value or use an EL Expression that evaluates to the reference area item's value. For example, to specify the minimum value for the reference area item for the line chart shown in [Figure 23-51](#), enter `#{row.value + 15}`.

- **X:** Specify the location on the x-axis where the reference line item or reference area item is to be rendered.

For charts with time data, this is the time stamp. For charts with a categorical axis, this is the index of the group, which starts at 0.

To specify the x-axis position for the line chart shown in [Figure 23-51](#), enter `#{row.series == 'Series 1' ? row.group:0}`.

10. To add additional reference line items, repeat Step 7 through Step 9 for each additional item.

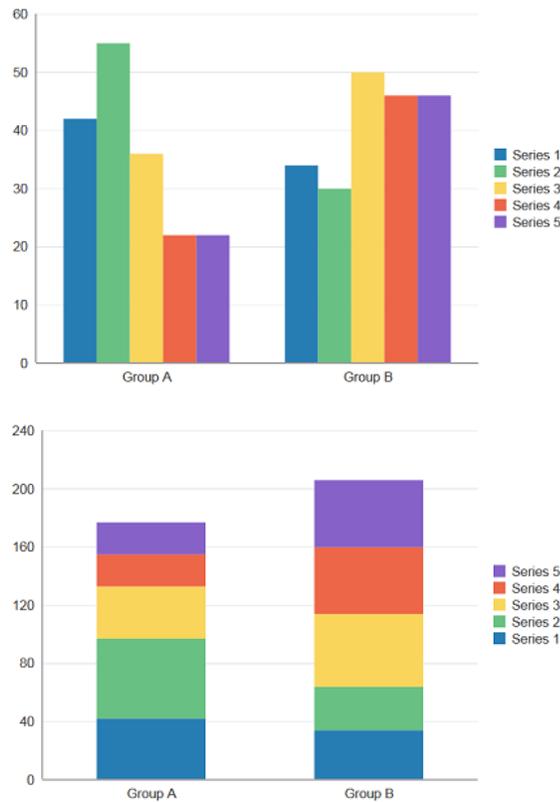
What You May Need to Know About Adding Reference Objects to Charts

The multi-segmented reference line or area is supported on bubble charts, scatter charts, and charts configured with a time axis.

How to Configure a Stacked Chart

Stacked charts show cumulative values across groups. Stacked charts typically contain two or more series which are aggregated into one bar, area, or line. For example, you might want to show the total sales of three products, grouped by city. In a clustered bar chart, the bar chart would display three bars, one for each product. In a stacked bar chart, the three bars would be aggregated into one bar for each city.

[Figure 23-52](#) shows the effect of stacking series in a bar chart. In this example, the five series in the clustered bar chart at the top of the figure are configured as a stacked bar chart at the bottom of the figure.

Figure 23-52 Stacked Bar Chart

If you chose a quick start layout that included a stacked display, then JDeveloper automatically stacked the series by group. Otherwise, you can enable stacking by setting the `stack` attribute to `on`.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure a stacked chart:

1. In the Structure window, right-click `dvt:typeChart` and choose **Go To Properties**.
2. In the Properties window, from the **Stack** attribute's dropdown list, select `on`.

Customizing Chart Series

You can customize the fill of all series in a chart by setting the `seriesEffect` attribute on the chart component.

You can customize the appearance of individual series in a chart by adding the `dvt:chartSeriesStyle` component to your chart. Depending upon the chart type, you can customize colors, markers, lines, fill effects and click actions. For combination charts, you can also specify which chart to display.

 **Note:**

Spark charts contain only a single series and do not support `seriesEffect` or `chartSeriesStyle`. To customize the spark chart, enter values for the `dvt:sparkChart` component in the Properties window. You can customize the series color and specify which markers are displayed. For more information, click **Component Help** in the Properties window.

How to Customize a Chart Series

To customize an individual series in a chart, add the `dvt:chartSeriesStyle` component and configure its properties in the Properties Window.

[Table 23-7](#) lists the attributes available on the `chartSeriesStyle` component and the chart types for which they apply.

Table 23-7 `dvt:chartSeriesStyle` Tags

Chart Tag	Description	Supported Chart Types
<code>action</code>	Method reference to an action.	area, bar, bubble, combination, line
<code>actionListener</code>	Method reference to an action listener.	area, bar, bubble, combination, line
<code>areaColor</code>	Specifies the color of an area series.	area, combination (with area)
<code>assignedToY2</code>	Specifies whether the series should be assigned to the Y2 axis.	area, bar, combination, line
<code>attributeChangeListener</code>	Method reference to an attribute change listener.	all
<code>borderColor</code>	Specifies the border color of the series.	all
<code>borderWidth</code>	Specifies the border width of the series	all
<code>color</code>	Specifies the color of the series.	all
<code>displayInLegend</code>	Specifies whether the series is displayed in the chart legend	all
<code>drilling</code>	Specifies whether this group can be drilled. Valid values are 'off' (default) and 'on'. When set to 'on', the existing <code>drillEvent</code> will be fired when the user clicks/taps on a the group label.	all
<code>id</code>	Specifies the id of the component.	all
<code>lineStyle</code>	Specifies the appearance of the line.	combination (with line), line

Table 23-7 (Cont.) dvt:chartSeriesStyle Tags

Chart Tag	Description	Supported Chart Types
lineType	Specifies the appearance of line connectors	area, bubble, scatter, combination (with line), line
lineWidth	Specifies the width of the line in pixels.	combination (with line), line
markerColor	Specifies the color of the data item markers, if different than the series color.	area, bubble, combination (with area or line), line, scatter
markerDisplayed	Specifies whether the data item markers are displayed.	area, bubble, combination (with area or line), line, scatter
markerShape	Specifies the shape of the data item markers.	area, bubble, combination (with area or line), line, scatter
markerSize	Specifies the size of the data item markers if displayed.	area, bubble, combination (with area or line), line, scatter
pattern	Specifies the pattern of the series.	all
rendered	Specifies whether the component is rendered.	all
series	Identifies the series for which the series style applies.	all
stackCategory	String value that specifies which series will be stacked together.	area, bar, combination, line
type	Specifies whether the series is displayed as an area, bar, line, or lineWithArea chart.	combination

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To customize a chart series:

1. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Facet seriesStamp**.
2. If you plan on customizing more than one series in your chart, right-click **f:facet - dataStamp** and choose **Insert Inside Facet series Stamp > Group**.

3. Right-click **af:group** or **f:facet seriesStamp** and choose **Insert Inside (Group or Facet seriesStamp) > Chart Series Style**.
4. To add additional series style elements, right-click **af:group** and choose **Insert Inside Group > Chart Series Style** for each additional element.
5. Right-click **dvt:chartSeriesStyle** and choose **Go to Properties**.
6. In the Properties window, enter values for the desired customization.
 - **Action**: Specify the method reference to use for the action attribute, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the method reference.
 - **ActionListener**: Specify the method reference to use for the action listener, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the method reference.
 - **AreaColor**: Specify the RGB value in hexadecimal notation to use for the area fill color, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter #000080 to display the area fill color in blue.
 - **AssignedToY2**: From the attribute's dropdown list, select `true` to customize the y2-axis.
 - **BorderColor**: Specify the RGB value in hexadecimal notation to use for the border color, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, enter #008000 to display the series border color in green.
 - **BorderWidth**: Specify the numeric value to use for the border width, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the numeric value.
For example, enter 4 to display the series border with a width of 4.
 - **Color**: Specify the RGB value in hexadecimal notation to use for the series color, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
 - **DisplayInLegend**: From the attribute's dropdown list, select `false` if you do not want to display the series in the legend. By default, this attribute is set to `true`.
 - **LineStyle**: From the attribute's dropdown list, select a line style. By default, this attribute is set to `solid` which will display a solid line. You can also set this to `dashed` or `dotted`.
 - **LineType**: From the attribute's dropdown list, select a line type. By default, this attribute is set to `auto`. You can also set this to `straight`, `curved`, `stepped`, `centeredStepped`, `segmented`, `centeredSegmented` or `none`.
 - **LineWidth**: Specify the width in pixels for the line.
 - **MarkerColor**: Specify the RGB value in hexadecimal notation to use for the series color, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
 - **MarkerDisplayed**: From the attribute's dropdown list, select `true` to display the line or area series marker. By default, this attribute is set to `false`.
 - **MarkerSize**: Enter a value in pixels for the marker size.

- **Pattern:** From the attribute's dropdown list, select a series pattern.
Available patterns include `smallChecker`, `smallCrosshatch`, `smallDiagonalLeft`, `smallDiagonalRight`, `smallDiamond`, `smallTriangle`, `largeChecker`, `largeCrosshatch`, `largeDiagonalLeft`, `largeDiagonalRight`, `largeDiamond`, and `largeTriangle`.
- **Series:** Enter the name of the series.
For example, to customize the first series in your chart, enter: `Series 1`.
If your application uses the Fusion technology stack and you are configuring a databound chart, the `Series` field will display the available bindings in the attribute's dropdown list. For more information about databound charts, see *Creating Databound Charts in Developing Fusion Web Applications with Oracle Application Development Framework*.
- **StackCategory:** Enter a string value to denote the **stackCategory** of the series. Series with the same **stackCategory** will be stacked together, allowing for some series to remain solo while others are stacked as desired. This attribute will only function if the **stack** attribute on the parent chart tag is set to `on`.
For example, in a chart with five series, add a string such as `stack1` to the **StackCategory** attribute of both `Series 1` and `Series 4` to stack them together and leave the others as individual bars.
- **Type:** From the attribute's dropdown list, select the chart type to use for the combination chart series.
Available choices include `area`, `bar`, `line`, or `lineWithArea`.

How to Configure Series Fill Effects on All Series in a Chart

To customize the series fill effects for all series in the chart, specify a value for the `seriesEffect` attribute.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure series fill effects for all series in a chart:

1. In the Structure window, right-click **`dvt:typeChart`** and choose **Go To Properties**.
2. In the Properties window, from the **SeriesEffect** attribute's dropdown list, select the desired fill effect.

By default, the `SeriesEffect` attribute is set to `gradient`. You can select `color` to render the chart without gradients or `pattern` to display each series in a different pattern.

If you want to specify specific patterns for a series, see [How to Customize a Chart Series](#).

Customizing Chart Groups

Chart groups support hierarchical labels for data item groups. These labels can be customized with styles, tooltips, and drilling.

You can customize chart group hierarchies by adding the `dvt:chartGroup` component to your chart. To create each level of hierarchy, you can nest an additional `dvt:chartGroup` tag. The hierarchy group labels can be unevenly distributed and support custom styling, including font face and size, text color and backgrounds.

Note:

Spark charts contain only a single series and do not support `chartGroup`. To customize the spark chart, enter values for the `dvt:sparkChart` component in the Properties window. You can customize the series color and specify which markers are displayed. For more information, click Component Help in the Properties window.

How to Customize a Chart Group

To customize an individual group in a chart, add the `dvt:chartGroup` component to the chart and configure its attribute values in the Properties Window.

[Table 23-8](#) lists the attributes available on the `chartGroup` tag.

Table 23-8 `dvt:chartGroup` Attributes

Attribute	Description
<code>drilling</code>	Specifies whether this group can be drilled. Valid values are 'off' (default) and 'on'. When set to 'on', the existing <code>drillEvent</code> will be fired when the user clicks/taps on a the group label.
<code>group</code>	Specifies the name of the group. The name will be used as a group label.
<code>groupId</code>	Specifies the group ID for matching the group with the <code>group</code> or <code>groupId</code> of the <code>ChartDataItem</code> . Required for leaf <code>chartGroup</code> elements when using hierarchical groups. The <code>groupId</code> must be unique for each <code>chartGroup</code> instance.
<code>labelStyle</code>	Specifies the label style for the group specified by the <code>group</code> attribute.
<code>rendered</code>	Specifies whether the component is rendered. Valid values are 'true' (default) and 'false'.
<code>shortDesc</code>	Specifies the text that appears as a tooltip upon hovering over the component. The tooltip is used for the group

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

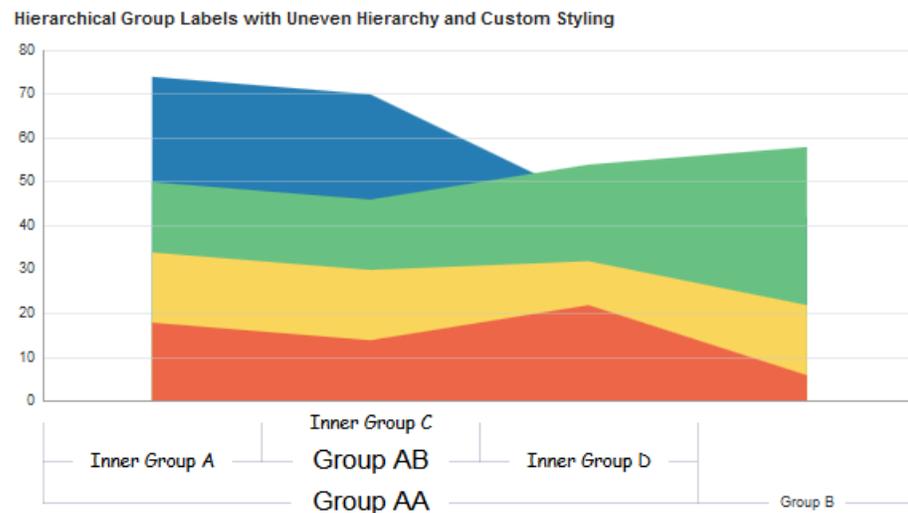
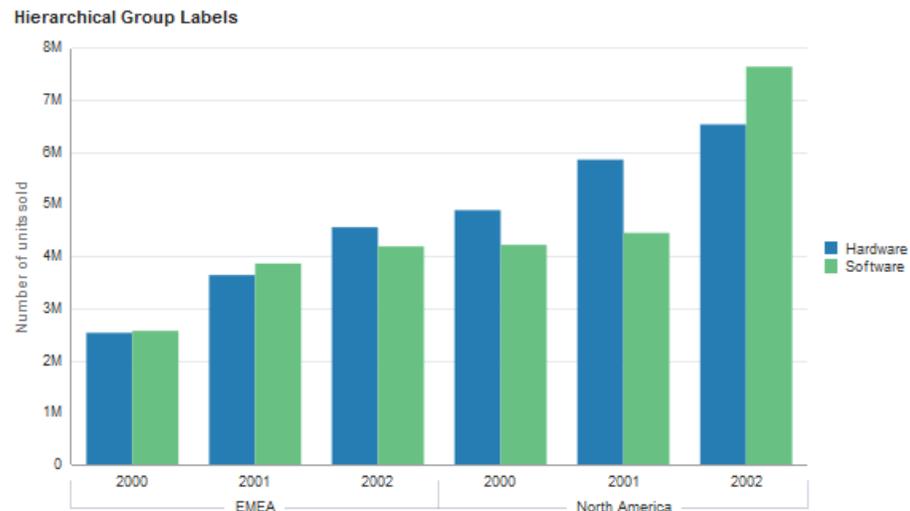
To customize a chart group:

1. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Facet groupStamp**.
2. If you plan on customizing more than one non—hierarchical group in your chart, right-click **f:facet - groupStamp** and choose **Insert Inside Facet group Stamp > Group**.
3. Right-click **af:group** or **f:facet groupStamp** and choose **Insert Inside (Group or Facet groupStamp) > Chart Group**.
4. To add additional group elements, right-click **af:group** and choose **Insert Inside Group > Chart Group** for each additional element.
5. Right-click **dvt:chartGroup** and choose **Go to Properties**.
6. In the Properties window, enter values for the desired customization.
 - **Group**: Enter a string value or choose **Expression Builder** from the attribute's dropdown menu to provide an expression that evaluates to the string that will be displayed as the group label.
 - **GroupId**: Enter a string value or choose **Expression Builder** from the attribute's dropdown menu to provide an expression that evaluates to a string that matches the corresponding value of the `ChartDataItem`.
 - **LabelStyle**: Enter a string value or choose **Expression Builder** from the attribute's dropdown menu to provide an expression that evaluates to the custom CSS markup for styling the group label.
 - **ShortDesc**: Enter a string value or choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to provide an expression that evaluates to the string that will be displayed as a tooltip upon hovering over the group.

How to Configure Hierarchical Labels Using Chart Groups

Chart groups support hierarchical nesting to represent multi-level grouping in data. [Figure 23-53](#) shows two charts with a hierarchical X-axis. The bar chart has even hierarchical distribution and the default styling on group labels. The area chart has uneven distribution and custom styles applied.

Figure 23-53 Hierarchical Group Labels in Chart Axis



The example below shows the code snippet on the JSF page for the bar and area charts with custom hierarchical labels, with the relevant code for the labels highlighted in bold font.

```
<dvt:barChart value="#{chartDataSource.brandData}" var="row" inlineStyle="width:
700px;height:400px" title="Hierarchical Group Labels">
  <f:facet name="dataStamp">
    <dvt:chartDataItem series="#{row.brand}" group=" "
groupId="#{row.region}_#{row.year}" value="#{row.value}"/>
  </f:facet>
  <f:facet name="groupStamp">
    <dvt:chartGroup group="#{row.region}" groupId="#{row.region}">
      <dvt:chartGroup group="#{row.year}"
groupId="#{row.region}_#{row.year}"/>
    </dvt:chartGroup>
  </f:facet>
  <dvt:chartYAxis title="Number of units sold"/>
  <dvt:chartLegend rendered="true"/>
</dvt:barChart>
```

```

<dvt:chartValueFormat type="group" tooltipLabel="Region" />
<dvt:chartValueFormat type="group" tooltipLabel="Year" />
<dvt:chartValueFormat type="series" tooltipLabel="Department" />
</dvt:barChart>

<dvt:areaChart value="{chartDataSource.defaultAreaData}" var="row"
inlineStyle="width:700px;height:400px" title="Hierarchical Group Labels with Uneven
Hierarchy and Custom Styling">
  <f:facet name="dataStamp">
    <dvt:chartDataItem series="{row.series}" group="{row.group}"
value="{row.value}"/>
  </f:facet>
  <f:facet name="groupStamp">
    <dvt:chartGroup group="Group AA" groupId="Group AA"
shortDesc="Outermost group" labelStyle="font-size: 20px; ">
      <dvt:chartGroup group="Inner #{row.group}"
groupId="{row.group}" shortDesc="Sub Level 1" labelStyle="font-size:
14px; font-family: Comic Sans MS" rendered="{row.group != 'Group B' and
row.group != 'Group C'}"/>
        <dvt:chartGroup group="Group AB" groupId="Group AB"
shortDesc="Sub Level 1" labelStyle="font-size: 20px; ">
          <dvt:chartGroup group="Inner #{row.group}"
groupId="{row.group}" shortDesc="Sub Level 2" labelStyle="font-size:
14px; font-family: Comic Sans MS" rendered="{row.group == 'Group C'}"/>
        </dvt:chartGroup>
      </dvt:chartGroup>
    </f:facet>
    <dvt:chartLegend rendered="false"/>
  </dvt:areaChart>

```

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. See [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To create a chart group hierarchy:

1. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Facet groupStamp**.
2. If you plan on customizing more than one non—hierarchical group in your chart, right-click **f:facet - groupStamp** and choose **Insert Inside Facet group Stamp > Group**.
3. Right-click **af:group** or **f:facet groupStamp** and choose **Insert Inside (Group or Facet groupStamp) > Chart Group**.
4. To create a nested group in your hierarchy, right-click the parent **dvt:chartGroup** and choose **Insert Inside Chart Group > Chart Group** for each additional level of nesting.
5. Repeat step 3 to create non-nested groups. Repeat step 4 to add more branch elements to your hierarchy.

6. Right-click a **dvt:chartGroup** tag you want to customize and choose **Go to Properties**.
 - Enter values in the `Group` and `GroupId` attributes to specify the data value for the group.
 - Enter values in appropriate attributes to configure the label styles.
7. Repeat step 6 for each **dvt:chartGroup** tag to provide data values and styles as required.

To learn about customizing label styles, see [How to Customize a Chart Group](#).

How to Configure the Pie Chart Other Slice

Use the Other slice to aggregate smaller data sets visually into one larger set for easier comparison, as shown in [Figure 23-36](#). To configure the Other slice, set a value for the `otherThreshold` attribute which specifies the percentage under which the slice would be aggregated into the Other slice. Optionally, you can set a value for the Other slice color.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure the pie chart Other slice:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
2. In the Properties window, expand the **Appearance** section.

By default, the `SeriesEffect` attribute is set to gradient. You can select color to render the chart without gradients or pattern to display each series in a different pattern.

If you want to specify specific patterns for a series, see [How to Customize a Chart Series](#).

3. In the **OtherThreshold** field, enter a value between 0 and 1 to set the percentage under which the slide will be aggregated.

For example, if you want to aggregate all slices whose values are less than two percent of the pie chart's total, enter `0.02`.

4. Optionally, in the **OtherColor** field, specify the RGB value in hexadecimal notation to use for the border color, or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.

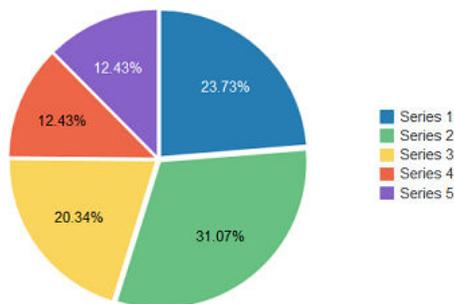
For example, enter `#008000` to display the Other slice in green.

How to Explode Pie Chart Slices

You can configure the slices of a pie chart so that each slice is separated from the other using the `sliceGaps` attribute of the `dvt:pieChart` tag or the `explode` attribute of the `dvt:pieDataItem` tag. You can specify a value between 1 and 10 for the distance between the slices.

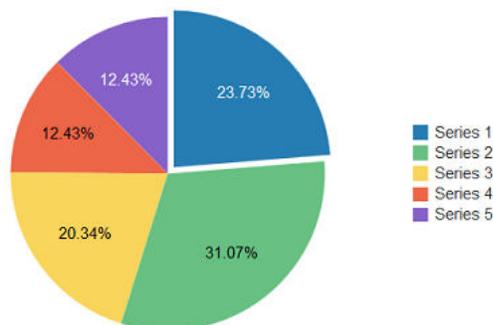
Figure 23-54 shows the pie chart displayed in Figure 23-41 configured for exploding slices. In this example, the `explode` attribute for the pie chart's data item is defined as 0.25.

Figure 23-54 Pie Chart Configured For Exploding Slices



You can also configure an individual slice to explode using the `explode` attribute. You can also configure a selected slice to automatically explode using the `SelectionEffect` attribute of the pie chart. Figure 23-55 shows the same pie chart configured to explode the Series 1 slice.

Figure 23-55 Pie Chart Configured With Single Exploding Slice



To configure an individual slice, you can use an EL expression to identify the slice and the value for the `explode` attribute. The individual slices in a pie chart are its series, and you can reference the series number in the EL expression.

For example, to set the `explode` attribute for the first series (slice) in the pie chart to 0.5, enter the following for the EL expression: `#{row.series == 'Series 1' ? 0.5 : 0}`. If you wanted to set the `explode` attribute for the third series to 0.25, you could enter the following for the EL expression: `#{row.series == 'Series 3' ? 0.25 : 0}`.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To explode pie slices:

1. In the Structure window, right-click **dvt:pieDataItem** and choose **Go To Properties**.
2. In the Properties window, expand the **Appearance** section.
3. To explode all slices in the pie chart, in the **Explode** field, enter a value between 0 and 1.
4. To explode a single slice in the pie chart, in the **Explode** field, enter an EL Expression that evaluates to the series name and explode value.

For example, to explode the first series in the pie chart shown in [Figure 23-55](#), enter the following in the **Explode** field: `#{row.series == 'Series 1' ? 0.5 : 0}`.

How to Configure Animation

To configure chart animation, add the `af:transition` tag as a child of the chart component and configure the trigger type and transition effect.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure chart animation:

1. In the Structure window, click the chart component.
2. In the Source editor, add the `af:transition` tag as a child of the highlighted chart component as shown in the following example.

```
<af:transition triggerType="display" transition="auto"/>
```

What You May Need to Know About Skinning and Customizing Chart Display Elements

Charts also support skinning to customize the color and font styles for the top level components as well as the axes, legend, series, marquee icon, labels, and plot area. You can also use skinning to define the animation duration and chart series effect.

The code below shows the skinning key for a chart configured to show patterns for its series fill effect.

```
af|dvt-chart
{
  -tr-series-effect: pattern;
}
```

For the complete list of chart skinning keys, see the *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces Skin Selectors*. For additional information about customizing your application using skinning and styles, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactive Features to Charts

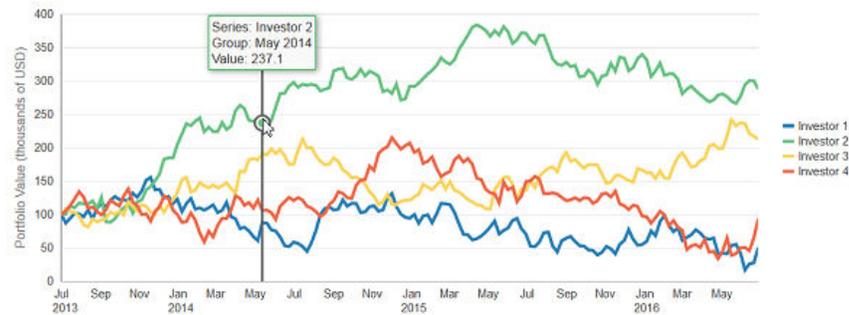
You can add a variety of interactive features to ADF DVT charts, including data cursors, hide and show behavior, hover behavior, selection support, popups, context menus, and zoom and scroll.

How to Add a Data Cursor

Add a data cursor to your chart to allow the user to focus more easily on data points. For charts where selection and other click interactivity is not enabled, the data cursor can be used to provide feedback for the closest data item to the mouse or touch gesture.

The data cursor is enabled by default for area and line charts on touch devices. To add the data cursor explicitly, set the `dataCursor` attribute to `on` and define the data cursor's behavior.

[Figure 23-56](#) shows the line chart displayed in [Figure 23-40](#) configured with a data cursor. In this example, the data cursor displays the series name, group value and series value. The box surrounding the data detail is displayed in the same color as the series fill color.

Figure 23-56 Line Chart With Data Cursor Showing Investor Detail

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure a chart data cursor:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
2. In the Properties window, expand the **Data** section.
3. From the **DataCursor** attribute's dropdown list, select `on` to enable the data cursor.
4. From the **DataCursorBehavior** attribute's dropdown list, select `smooth` or `snap` to specify the data cursor behavior.

By default, the data cursor's behavior is set to `auto`, and the data cursor moves smoothly for line and area charts, and it snaps for other chart types. You can set this to `smooth` or `snap` to set the behavior explicitly.

Note:

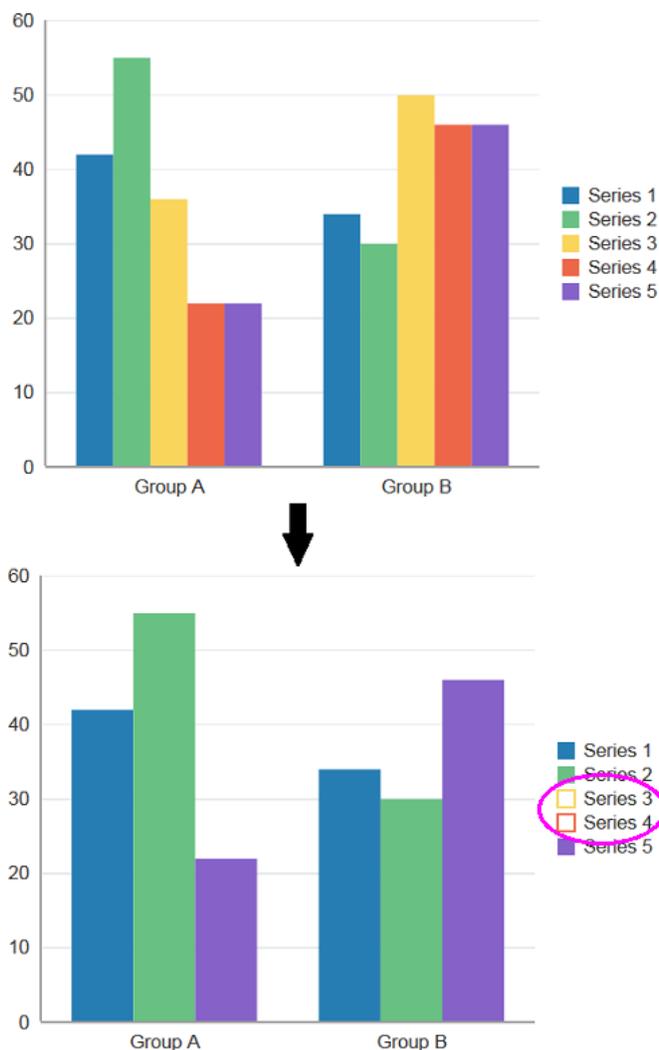
The content displayed in the data cursor's tooltip is determined by the `dvt:chartDataItem` or `dvt:pieDataItem` component's `shortDesc` attribute. You can edit this value in the Properties window for `dvt:chartDataItem` or `dvt:pieDataItem`.

How to Configure Hide and Show Behavior

To configure hide and show behavior, which permits the user to click on a legend series item to hide or show a series item, configure the chart's `hideAndShowBehavior` attribute.

Figure 23-57 shows a bar chart configured for hide and show behavior support. The user can click multiple legend items to hide the respective series items, and click again to show them again. The legend items have a white fill to indicate which series are hidden.

Figure 23-57 Bar Chart Configured for Hide and Show Behavior



Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure hide and show behavior:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
2. In the Properties window, expand the **Behavior** section.
3. From the **HideAndShowBehavior** attribute's dropdown list, select `withRescale` or `withoutRescale` to specify the data cursor behavior.

If you select `withoutRescale`, the chart will not rescale the axes to fit the data. Select `withRescale` if you want the chart to rescale the axes.

How to Configure Legend and Marker Dimming

To configure legend and marker dimming, set the chart's `hoverBehavior` attribute to `dim`. As the user hovers over each series, the remaining series dim from view.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure legend and marker dimming:

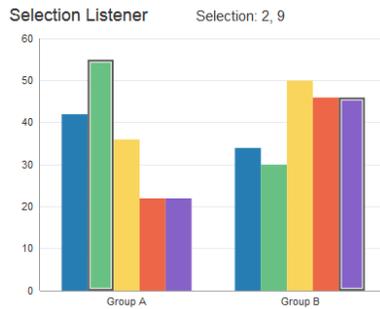
1. In the Structure window, right-click **dvt:typeChart** and choose **Go To Properties**.
2. In the Properties window, expand the **Behavior** section.
3. From the **HoverBehavior** attribute's dropdown list, select `dim`.

How to Configure Selection Support

Charts can be enabled for single or multiple selection of data markers. Enabling selection is required for popups and context menus and for responding programmatically to user clicks on the data markers. To enable selection support, set the chart's `dataSelection` attribute to `single` or `multiple`.

After you have enabled selection support, you can specify a selection listener that will respond to user clicks on the chart.

[Figure 23-58](#) shows a bar chart configured for multiple selection support. The user can click to select a single bar or use Ctrl-Click to select multiple bars. The bars highlight to show which bars are selected, and a message is displayed at the top of the page indicating which row key was selected.

Figure 23-58 Bar Chart Configured for Multiple Selection

The example below shows the code on the JSF page that defines the bar chart and selection listener, with the selection code highlighted.

```
<af:group id="g1">
  <af:outputText inlineStyle="font-size:large;" value="Selection Listener" id="ot1"/>
  <af:spacer width="50px"/>
  <af:outputText partialTriggers="chartSelect"
    value="#{chartDataSource.selectionState}"
    inlineStyle="font-size:larger;" id="ot2"/>
  <af:panelGroupLayout id="pg1" layout="horizontal">
    <dvt:barChart id="chartSelect" value="#{chartDataSource.defaultBarData}"
      var="row" dataSelection="multiple"
      selectionListener="#{chartDataSource.selectionListener}">
      <dvt:chartLegend id="leg1" rendered="false"/>
      <f:facet name="dataStamp">
        <dvt:chartDataItem id="cd1" value="#{row.value}" group="#{row.group}"
          series="#{row.series}"/>
      </f:facet>
    </dvt:barChart>
  </af:panelGroupLayout>
</af:group>
```

In this example, the code that defines the bar chart is included in a method named `defaultBarData()`, and the selection listener is defined in a method named `selectionListener()`.

The code below shows the `defaultBarData()` and `selectionListener()` methods. The methods and imports were added to the `chartDataSource` class created in the first code sample in [How to Add Data to Area, Bar, Combination, and Line Charts](#).

```
// Additional imports needed by selection listener
import oracle.adf.view.faces.bi.component.chart.UIChartBase;import
org.apache.myfaces.trinidad.event.SelectionEvent;
import org.apache.myfaces.trinidad.model.RowKeySet;

// Bar Chart data
public CollectionModel getDefaultBarData() { List<ChartDataItem> dataItems = new
ArrayList<ChartDataItem>(); dataItems.add(new ChartDataItem("Series 1", "Group A",
42)); dataItems.add(new ChartDataItem("Series 1", "Group B", 34));
dataItems.add(new ChartDataItem("Series 2", "Group A", 55)); dataItems.add(new
ChartDataItem("Series 2", "Group B", 30)); dataItems.add(new ChartDataItem("Series
3", "Group A", 36)); dataItems.add(new ChartDataItem("Series 3", "Group B", 50));
dataItems.add(new ChartDataItem("Series 4", "Group A", 22)); dataItems.add(new
ChartDataItem("Series 4", "Group B", 46)); dataItems.add(new ChartDataItem("Series
5", "Group A", 22)); dataItems.add(new ChartDataItem("Series 5", "Group B", 46));
```

```
return ModelUtils.toCollectionModel(dataItems);
}

// Selection state
private String m_selection = "No Nodes Selected";
public String getSelectionState() {
    return m_selection;
}

// Selection Listener
public void selectionListener(SelectionEvent event) {
    UIChartBase chart = (UIChartBase) event.getComponent();
    RowKeySet rowKeySet = chart.getSelectedRowKeys();
    if(rowKeySet != null && rowKeySet.size() > 0) {
        StringBuilder sb = new StringBuilder("Selection: ");
        for(Object rowKey : rowKeySet) {
            sb.append(rowKey).append(", ");
        }
        // Remove the trailing comma and set the selection string
        sb.setLength(sb.length()-2);
        m_selection = sb.toString();
    }
    else
        m_selection = "No Nodes Selected";
}
```

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure chart selection support:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go to Properties**.
2. In the Properties window, expand the **Data** section.
3. From the **DataSelection** dropdown list, select `single` or `multiple` to enable selection support.
4. Optionally, to enable a selection listener, do the following.
 - a. Create the method or methods that define the selection listener, and add it to the chart's managed bean.

If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
 - b. In the Properties window, in the **SelectionListener** field, enter the name of the selection listener.

For example, for a managed bean named `chartDataSource` and a method named `selectionListener()`, enter the following in the **SelectionListener** field: `#{chartDataSource.selectionListener}`.

You can also choose **Edit** from the **SelectionListener** attribute's dropdown menu to select a managed bean and method in the Edit Property: Selection Listener dialog, or choose **Expression Builder** to enter an expression that returns the selection listener.

5. Configure any additional elements as needed.

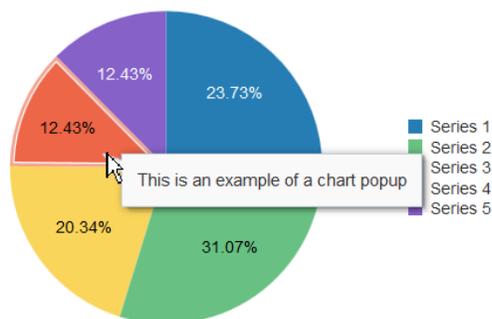
For example, to duplicate the multiple selection example in this section, add the `af:outputText` components shown in the first code sample in [How to Configure Selection Support](#).

How to Configure Popups and Context Menus

The process to add a popup or context menu is essentially the same. Add the `af:showPopupBehavior` tag as a child of one of the chart's data items, define the trigger type as `click` for popup menus or `contextMenu` for context menus, and add an `af:popup` containing the desired behavior to the page.

[Figure 23-59](#) shows the pie chart displayed in [Figure 23-41](#) configured for popup support. If the user clicks one of the pie slices, a note window pops up with a message.

Figure 23-59 Pie Chart Configured With a Note Window Popup



The example below shows the code on the page for the popup menu shown in [Figure 23-59](#). In this example, the `af:showPopupBehavior` component uses the `popupId` to reference the `af:popup` component. The `af:popup` component is configured with the `af:noteWindow` component which is configured to display a simple message in the `af:outputFormatted` component. The `triggerType` of the `af:showPopupBehavior` tag is set to `click`, and the note window will launch when the user clicks one of the pie chart's slices.

```
<af:group id="g1">
  <dvt:pieChart id="chart1" value="#{chartDataSource.defaultPieData}"
    var="row" dataSelection="single">
    <dvt:chartLegend id="leg1" rendered="true"/>
    <dvt:pieDataItem label="#{row.series}" value="#{row.value}" id="pdil">
      <af:showPopupBehavior popupId="::noteWindowPopup" triggerType="click"
        align="afterStart"/>
    </dvt:pieDataItem>
  </dvt:pieChart>
  <af:popup childCreation="deferred" autoCancel="disabled"
    id="noteWindowPopup" clientComponent="true"
```

```

        launcherVar="source" eventContext="launcher">
    <af:noteWindow id="nw1">
        <af:outputFormatted value="This is an example of a chart popup"
id="of1"/>
    </af:noteWindow>
</af:popup>
</af:group>

```

You can change the popup to a context menu by simply changing the trigger type for the `af:showPopupBehavior` component to `contextMenu` as shown in the following code snippet:

```
<af:showPopupBehavior popupId="::noteWindowPopup" triggerType="contextMenu"/>
```

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

Add a popup component to your page. For help with configuring the `af:popup` component, see [Using Popup Dialogs, Menus, and Windows](#).

To add a popup or context menu to a chart:

1. In the Structure window, right-click **dvt:typeChart** and choose **Insert Inside Chart > Show Popup Behavior**.
2. Right-click **af:showPopupBehavior** and choose **Go to Properties**.
3. In the Properties window, enter values for the following:
 - **PopupId**: Specify the ID of the `af:popup` component.
 - **TriggerType**: For popup menus, enter `click`. For context menus, enter `contextMenu`.

Optionally, set values for **Align**, **AlignId**, and **Disabled**. Click **Component Help** for more information about the `af:showPopupBehavior` component.

4. In the Structure window, right-click **dvt:typeChart** and choose **Go to Properties**.
5. In the Properties window, expand the **Data** section.
6. From the **DataSelection** dropdown list, select `single` or `multiple` to enable selection support.

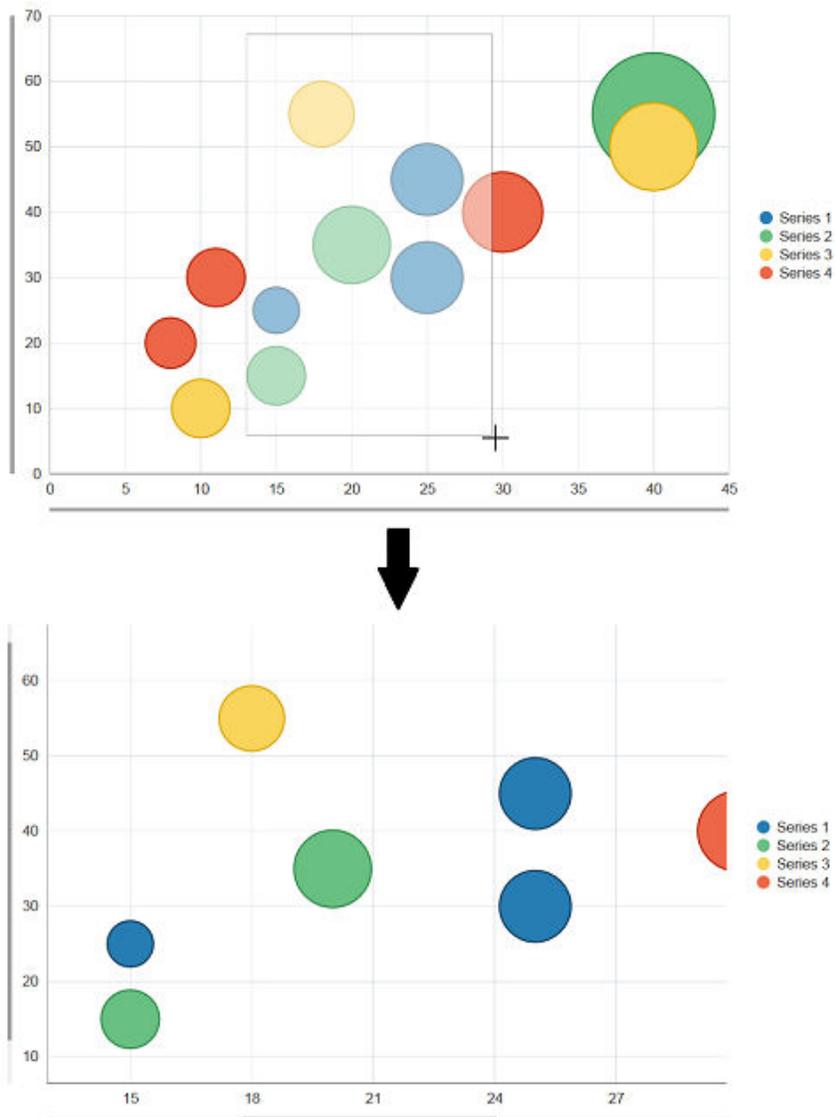
How to Configure Chart Zoom and Scroll

You can configure your chart to include marquee zoom and scroll which permits the user to focus on an area of the chart or scroll through the data using the mouse. This feature can be useful for large data sets.

[Figure 23-60](#) shows a bubble chart configured for marquee zoom and scroll. The user can select an area on the chart and then release the mouse button to zoom in on the

selected area. The user can also scroll the mouse wheel upward to zoom in on chart data. To restore the chart to its original display, the user can scroll the mouse wheel downward.

Figure 23-60 Bubble Chart Configured for Marquee Zoom and Scroll



To configure marquee zoom and scroll, set a value for the chart's `zoomAndScroll` attribute.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To configure marquee zoom and scroll:

1. In the Structure window, right-click **dvt:typeChart** and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section.
3. From the **ZoomAndScroll** dropdown list, select the desired zoom and scroll behavior. By default, this attribute is set to `off`. Available options include:

- `delayed`: Specifies that the chart update will wait until the zoom or scroll action is done. Both zoom and scroll will be enabled.

Specify a delay if the chart display is slow to render.

- `delayedScrollOnly`: Specifies that the chart update will wait until the scroll action is done. The marquee zoom icon will not be displayed.
- `live`: Specifies that the chart will be updated continuously as it is being manipulated. Both zoom and scroll will be enabled.
- `liveScrollOnly`: Specifies that the chart will be updated continuously as it is being manipulated. The marquee zoom icon will not be displayed.

You can also choose **Expression Builder** from the **ZoomAndScroll** attribute's dropdown menu to enter an expression that returns the zoom and scroll behavior.

4. Optionally, to enable a selection listener, do the following.
 - a. Create the method or methods that define the selection listener, and add it to the chart's managed bean.

If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
 - b. In the Properties window, in the **SelectionListener** field, enter the name of the selection listener.

For example, for a managed bean named `chartDataSource` and a method named `selectionListener()`, enter the following in the **SelectionListener** field: `#{chartDataSource.selectionListener}`.

You can also choose **Edit** from the **SelectionListener** attribute's dropdown menu to select a managed bean and method in the Edit Property: Selection Listener dialog, or choose **Expression Builder** to enter an expression that returns the selection listener.

5. Configure any additional elements as needed.

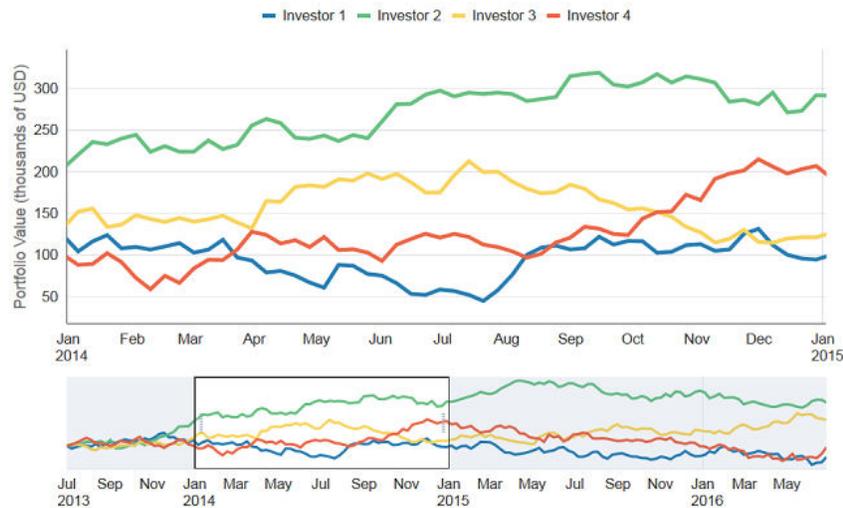
For example, to duplicate the multiple selection example in this section, add the `af:outputText` components shown in the first code sample in [How to Configure Selection Support](#).

How to Configure Chart Overview Window

You can add a viewport to the chart which uses a small form factor to display the entire data set, and then configure the original line chart to display a subset of the data.

Figure 23-61 shows the line chart in Figure 23-40 configured with an overview port. When the user zooms in on a chart area, the viewport changes to match the user's selection. The user can stretch or shrink the viewport dragging the handles on each side of the viewport.

Figure 23-61 Line Chart Configured With Overview



To configure an overview window, add the overview facet and overview window to the chart, and configure the chart's axis for the overview range.

Before you begin:

It may be helpful to have an understanding of how chart attributes and child tags can affect functionality. For more information about configuring charts, see [Configuring Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Chart Components](#).

Add a chart to your page. For help with adding a chart to a page, see [How to Add a Chart to a Page](#).

To add an overview window to your chart:

1. In the Structure window, right-click `dvt:typeChart` and choose **Insert Inside Chart > Facet overview**.
2. Right-click `f:facet - overview` and choose **Insert Inside Facet overview > Overview**.
3. Right-click `dvt:overview` and choose **Go to Properties**.
4. In the Properties window, enter values to style the overview window as desired.
For example, you can specify an inline style or style class to use for the overview window. For help with the overview window properties, click **Component Help**.
5. If your chart does not include an x-axis, in the Structure window, right-click `dvt:typeChart` and choose **Insert Inside Chart > Chart XAxis**.

6. Right-click **dvt:chartXAxis** and choose **Go to Properties**.
7. In the Properties window, expand the **Viewport** section.
8. From the **Viewport** attribute's dropdown list, enter values for the following as needed to set the viewport's range.

- **ViewportStartGroup**: Specifies the start group of the current viewport. This attribute only applies to charts with a group or time x axis. If not specified, the default start group is the first group in the data set.

For example, if your chart's groups consist of city data, you could enter the city name for the **ViewportStartGroup**: London.

You can also choose **Expression Builder** from the attribute's dropdown menu to enter an expression that returns the viewport's start group.

- **ViewportEndGroup**: Specifies the end group of the current viewport. This attribute only applies to charts with a group or time x axis. If not specified, the default end group is the last group in the data set.
- **ViewportMinimum**: Specifies the minimum x-axis coordinate of the current viewport for zoom and scroll.

For a group axis, the group index will be treated as the x-axis coordinate. For a time axis, the time stamp of the group will be treated as the x-axis coordinate. If both **viewportStartGroup** and **viewportMinimum** are specified, then **viewportStartGroup** takes precedence. If not specified, this value will be the axis minimum.

You can also choose **Expression Builder** from the attribute's dropdown menu to enter an expression that returns the viewport minimum. For the line chart displayed in [Figure 23-61](#), a method is added to the `chartDataSource` managed bean to return the viewport minimum.

The code below shows a sample method that returns the viewport minimum. In this example, the chart is configured with a viewport minimum of January 1, 2016.

```
// Add these imports to your bean
import java.util.Date;
import java.util.GregorianCalendar;

public Date getStockViewportMinimum() {
    return new GregorianCalendar(2016, 0, 1).getTime();
}
```

To use this method for your chart, enter the following for the **ViewportMinimum**: `#{chartDataSource.stockViewportMinimum}`.

- **ViewportMaximum**: Specifies the maximum x-axis coordinate of the current viewport for zoom and scroll.

You can also choose **Expression Builder** from the **ViewportMaximum** attribute's dropdown menu to enter an expression that returns the zoom and scroll behavior.

Using Picto Chart Components

This chapter describes how to use the ADF Data Visualization Picto Chart component to display data in picto charts using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of these components.

If your application uses the Fusion technology stack, you can use data controls to create picto charts. For more information, see "Creating Databound Picto Charts" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- [About the Picto Chart Component](#)
- [Using the Picto Chart Component](#)

About the Picto Chart Component

The ADF DVT Picto Chart components use icons to visualize an absolute number, or the relative sizes of the different parts of a population. Picto Charts are extensively used in infographics as a more interesting and effective way to present numerical information than traditional tables and lists.

Picto Charts are intended to be used in the default flowing layout. This makes the tool versatile since it can be easily used to visually enhance or support information in text or other ADF Data Visualization components. For example, the different slices of a funnel chart can be represented in absolute numbers as different parts of a population in a picto chart.

Picto Chart Use Cases and Examples

Picto Charts are a versatile tool that can be used along with other components to display data in a visually appealing manner. There are several use cases.

Picto charts can be used to enhance statements about absolute numbers and act as a visual aid. [Figure 24-1](#) shows picto charts reflecting the absolute value mentioned in the accompanying statements.

Figure 24-1 Picto Charts with Absolute Values



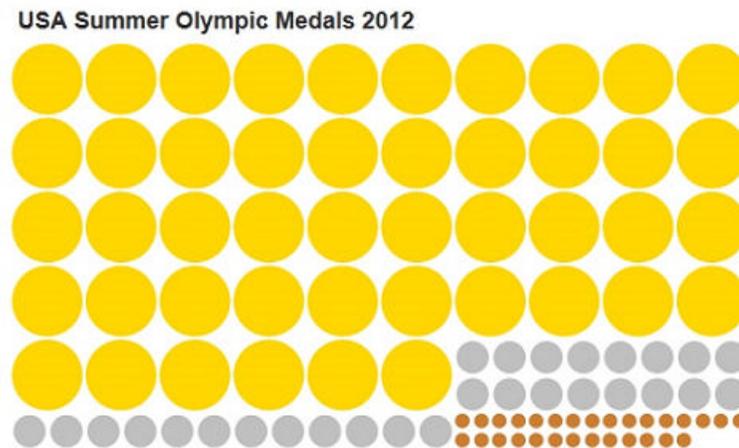
Picto Charts can be used to highlight a portion of a population as a comparison. This works best when the ratio is small. [Figure 24-2](#) shows a single picto chart configured to highlight three out of twenty human figures to illustrate the point made in the accompanying statement.

Figure 24-2 Simple Picto Chart to illustrate a Part of a Population



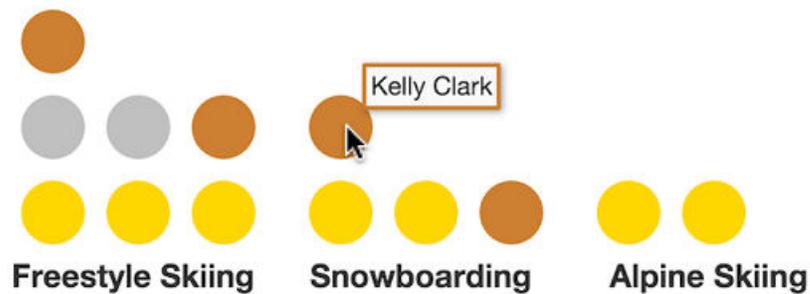
Another use case of picto charts is to create a comparison between parts of a population by highlighting the distribution of parts. For example, [Figure 24-3](#) shows three picto charts together in the default flow layout, each configured to show the number of medals won by the USA at the 2012 Summer Olympics. The three picto charts have been configured to correlate the size of the circle with the rank of the medal.

Figure 24-3 Picto Chart with varying sizes of items



Picto charts may also be used a set of singletons in order to display unique information for each data point. [Figure 24-4](#) shows a collection of picto charts to represent USA's medal count in the 2014 Winter Olympics. The `count` attribute for each picto chart is set to 1 and the tooltip for each picto chart is configured to show the name of the athlete who won that medal.

Figure 24-4 A collection of Singleton Picto Charts with Unique Information



Multiple picto charts can be arranged within a table to highlight business data. [Figure 24-5](#) shows a table with sales figures of Apple products in the years 2011 and 2012. Each square represents 10 million units.

Figure 24-5 Picto Charts arranged within Table

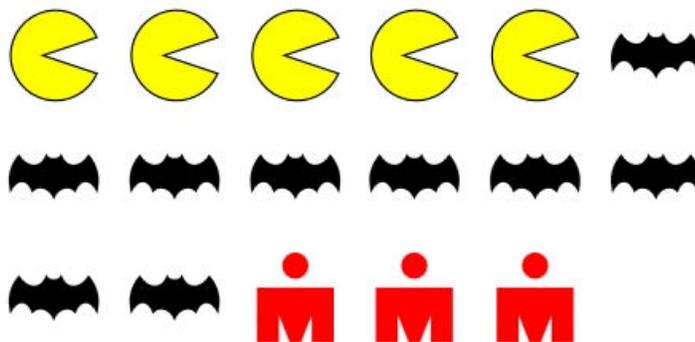
Product	2011	2012	Total (million units)
iPhone			132
iPad			17
iPod			77
Mac			35

End User and Presentation Features of Picto Charts

Picto Charts have several end user and configurable presentation features.

Picto Charts support custom shapes via the `source` attribute. [Figure 24-6](#) shows a sample picto chart for a survey on preferred iconic character of choice. Three custom shapes have been provided to indicate the survey options and results.

Figure 24-6 Picto Chart with Custom Shapes



Picto Charts have flowing layouts, which means that data items are rendered next to each other in the direction provided, given that the container does not have a set width and height.

Picto Chart data items can be rendered in different combinations of directions. Picto charts may be laid out in a horizontal flow or a vertical flow depending on the value of the `layout` attribute, and the data items can be rendered from any of the four corners of the container. [Figure 24-7](#) shows how data items are laid out in a horizontal layout depending on the value of the `layoutOrigin` attribute.

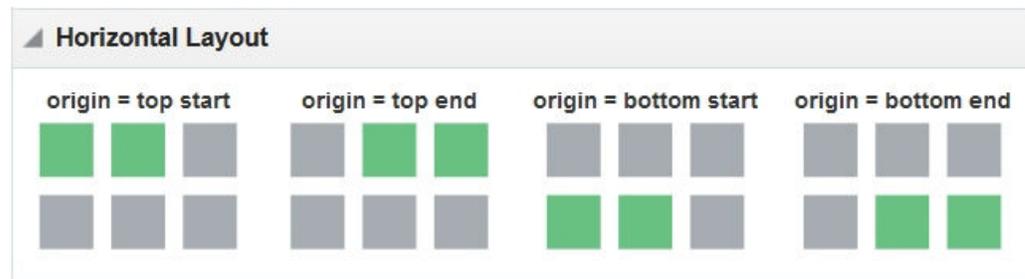
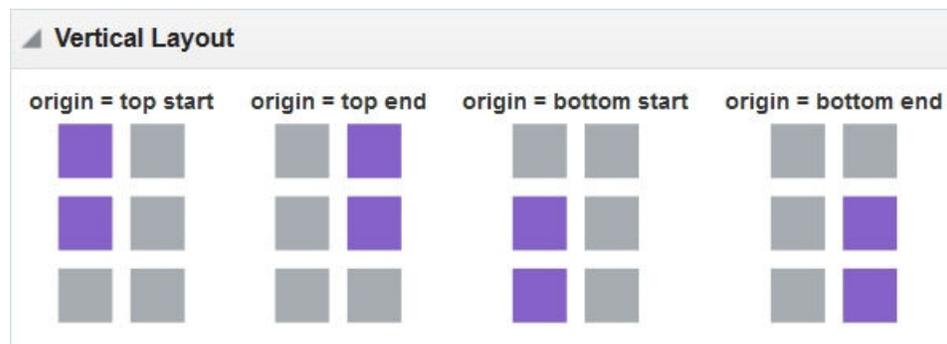
Figure 24-7 Layout Origin for Picto Chart Data Items in a Horizontal Layout

Figure 24-8 shows how data items are laid out in a vertical layout depending on the value of the `layoutOrigin` attribute.

Figure 24-8 Layout Origin for Picto Chart Data Items in a Vertical Layout

Additional Functionality for Picto Chart Components

You may find it helpful to understand other ADF Faces features before you implement your picto chart. Additionally, once you have added a picto chart to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that picto chart components can use:

- Client-side framework: DVT components are rendered on the client when the browser supports it. You can respond to events on the client using the `af:clientListener` tag. For more information, see [Listening for Client Events](#).
- Partial page rendering: You may want a picto chart to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: When enabled, users can change the way the picto chart displays at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: You can make your picto chart components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For

additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).

- Skins and styles: You can customize the appearance of picto chart components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound picto charts based on how your ADF Business Components are configured. For more information, see [Creating Databound Picto Charts in *Developing Fusion Web Applications with Oracle Application Development Framework*](#).

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Picto Chart Component

To use the ADF DVT Picto Chart component, add the picto chart to a page using the Component Palette window. Then define the data for the picto chart and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Picto Chart Data Requirements

The picto chart in its most basic form only requires a single numeric value.

The picto chart uses the value specified in the `count` attribute to display an absolute representation of the number. To represent a portion of a population, two `pictoChartItem` can be used, with two `count` attributes specified to represent the active portion and inactive portion of the population. Similarly, to represent multiple parts of a population, additional `pictoChartItem` may be used and styled to show each part in a unique manner.

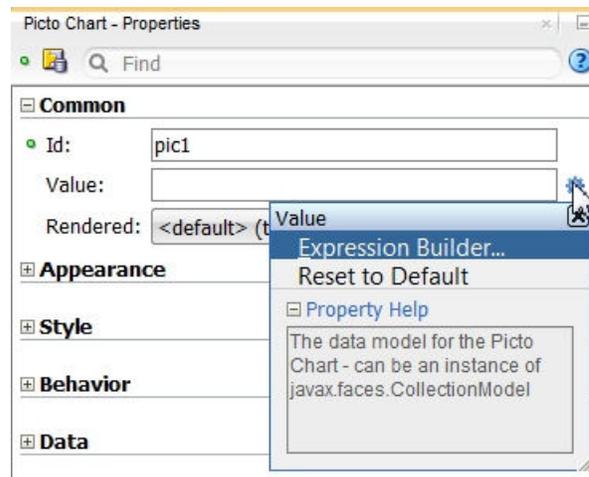
Data of type `int`, `long` or `double` may be used, since picto charts support fractional values. Data may be provided as an absolute entity or mapped to a data control.

How to Add a Picto Chart to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a picto chart to a JSF page. When you drag and drop a picto chart component onto the page, the picto chart is added to your page, and you can use the Properties window to specify data values and configure additional display attributes.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. The figure shows the dropdown menu for a picto chart `value` attribute.

Figure 24-9 Picto Chart Value Attribute Dropdown Menu



Note:

If your application uses the Fusion technology stack, then you can use data controls to create a Picto Chart and the binding will be done for you. For more information, see the "Creating Databound Picto Charts" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how picto chart attributes and picto chart child tags can affect functionality. For more information, see [Configuring Picto Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Picto Chart Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).

2. Create a view page as described in [Creating a View Page](#).

To add a picto chart to a page:

1. In the ADF Data Visualization page of the Components window, drag and drop a **Picto Chart** from the Common panel to the page.
2. In the Properties window, view the attributes for the Picto Chart. Use the **Help** button or press F1 to display the complete tag documentation for the `pictoChart` component.
3. In the Structure window, expand the **dvt:pictoChart** element to find the **dvt:pictoChartItem** element. View and modify the attributes for the **dvt:pictoChartItem** in the Properties window.
4. To create complex picto charts, right-click the **dvt:pictoChart** element and for each part of the population you wish to add, choose **Insert Inside Picto Chart > Picto Chart Item**.

What Happens When You Add a Picto Chart to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a picto chart from the Components window onto a JSF page.

The generated code is:

```
<dvt:pictoChart id="pic1">
  <dvt:pictoChartItem id="pi1"/>
</dvt:pictoChart>
```

After inserting a picto chart component into the page, you can use the visual editor or Properties window to add data or customize picto chart features. For information about setting component attributes, see [How to Set Component Attributes](#).

If you choose to bind the data to a data control when creating the picto chart, JDeveloper generates code based on the data model. For more information, see the [Creating Databound Picto Charts](#) section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Configuring Picto Charts

The Picto Chart component has configurable attributes and child components that you can add or modify to customize the display or behavior of the picto chart.

The prefix `dvt:` occurs at the beginning of the name of each picto chart component, indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library. You can configure Picto Chart child components, attributes, and supported facets in the following areas:

- **Picto Chart** (`dvt:pictoChart`): Wraps the picto chart child tags. Configure the following attributes to control the picto chart display:
 - `layoutRowCount`: The number of rows that the picto chart has.
 - `layoutColumnCount`: The number of columns that the picto chart has.
 - `layout`: The layout of the picto chart. May be horizontal or vertical.
 - `layoutOrigin`: The location where the first `pictoChartItem` will be rendered.

- `rowHeight`: The height of a row in pixels. Only applicable to picto charts using a flowing layout.
- `columnWidth`: The width of a row in pixels. Only applicable to picto charts using a flowing layout.
- Picto Chart data item (`dvt:pictoChartItem`): Use to define the properties for a portion of the represented population. Configure the following attributes to control the picto chart item display:

Table 24-1 Picto Chart Child Component Attributes

Child Tag Attribute	Description
<code>borderColor</code>	The border color of the icon. Does not apply if custom image is specified.
<code>borderWidth</code>	The border width of the icon. Does not apply if custom image is specified.
<code>color</code>	The color of the icon. Does not apply if custom image is specified.
<code>columnSpan</code>	The number of columns each icon spans. Used for creating a picto chart with mixed icon sizes. Defaults to 1.
<code>count</code>	The item count. Supports fractional values. Defaults to 1.
<code>name</code>	The name of the item. Used for default tooltip and accessibility.
<code>rowSpan</code>	The number of rows each icon spans. Used for creating a picto chart with mixed icon sizes. Defaults to 1.
<code>shape</code>	The shape of the icon. Does not apply if custom image is specified. Supported values are <code>circle</code> , <code>diamond</code> , <code>human</code> , <code>plus</code> , <code>rectangle</code> , <code>square</code> , <code>star</code> , <code>triangleUp</code> and <code>triangleDown</code> .
<code>source</code>	The image URI of the custom icon. If specified, it takes precedence over shape .
<code>sourceHover</code>	The optional URI for the hover state. if not specified, the source image will be used.
<code>sourceHoverSelected</code>	The optional URI for the hover selected state. if not specified, the source image will be used.
<code>sourceSelected</code>	The optional URI for the selected state. if not specified, the source image will be used.

- Picto Chart facets: Picto charts do not support facets.

Using Gauge Components

This chapter describes how to use the ADF Data Visualization dial, LED, rating, and status meter gauge components using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the Fusion technology stack, you can use data controls to create gauges. For more information, see "Creating Databound Gauges" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Gauge Component](#)
- [Using the Gauge Component](#)
- [Customizing Gauge Display Elements](#)
- [Adding Interactivity to Gauges](#)

About the Gauge Component

ADF DVT Gauges are measuring instruments for indicating a quantity such as sales, stock levels, temperature, or speed. Gauges typically display a single data value, often more effectively than charts.

Using thresholds, gauges can show state information such as acceptable or unacceptable ranges using color. For example, a gauge value axis might show ranges colored red, yellow, and green to represent low, medium, and high states. When you need to compare many data values at a glance, multiple gauges can be shown inside table cells.

Best Practice Tip:

When multiple data values, such as the text values of thresholds and the current value are required, a list or table component may be a better choice than a gauge.

The gauge component supports four categories of gauge types: dial, LED, rating, and status meter.

Gauge Component Use Cases and Examples

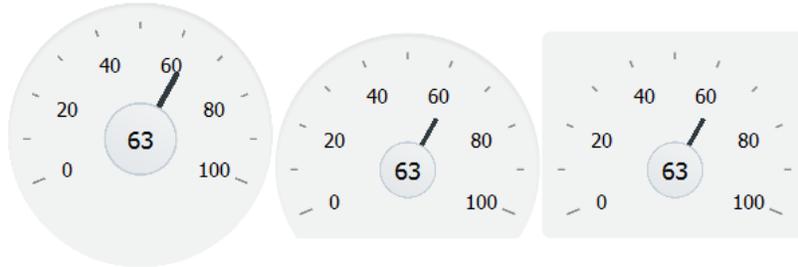
Gauges are typically used to display a single data point. The following types of gauges are supported by the gauge component:

- **Dial:** Displays a metric value plotted on a circular axis. The gauge's background attribute determines whether the gauge's background is displayed as a rectangle,

circle, or semicircle. An indicator points to the dial gauge's metric value on the axis.

Figure 25-1 shows three dial gauges with backgrounds set to full circle, partial circle, and rectangle. In all three examples, the gauge's metric value is 63.

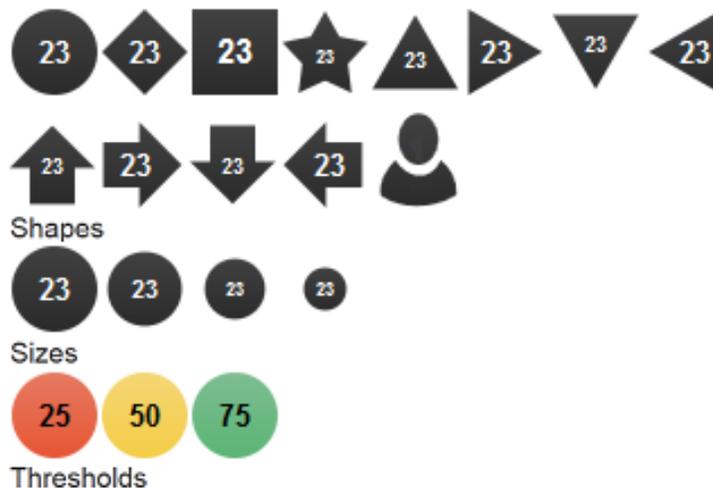
Figure 25-1 Dial Gauge Examples



- LED: Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of shapes are available for LED gauges, including round and rectangular shapes that use color to indicate status, and triangles or arrows that point up, left, right, or down indicate good (up), fair (left- or right-pointing), or poor (down) states in addition to the color indicator. A human shape is also available.

Figure 25-2 shows LED gauges configured with a variety of shapes, sizes, and thresholds.

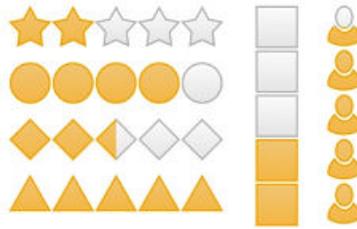
Figure 25-2 LED Gauge Examples



- Rating: Displays and optionally accepts input for a metric value. This gauge is typically used to show ratings for products or services, such as the star rating for a movie.

Figure 25-3 shows six rating gauges configured with star, circle, diamond, rectangle, triangle, and human shapes. Rating gauges may be changed to have either a horizontal or vertical orientation.

Figure 25-3 Rating Gauge Examples



- **Status Meter:** Displays the metric value on a horizontal, circular or vertical axis. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Optionally, status meters can display colors to indicate where the metric value falls within predefined thresholds.

Figure 25-4 shows examples of status meter gauges configured as horizontal, circular and vertical status meters. The gauges are configured to use thresholds that display color to indicate whether the gauge's value falls within an acceptable range.

Figure 25-4 Status Meter Gauge Examples



The figure shows an example of a circular status meter gauge configured to display a portion of the plot area. This is useful to depict key performance indicators (KPI) in a functional and visually intriguing manner.



End User and Presentation Features of Gauge Components

ADF Data Visualization gauge components provide a range of presentation features, such as shape variations, threshold display, visual effects, animation, and customizable color and label styles.

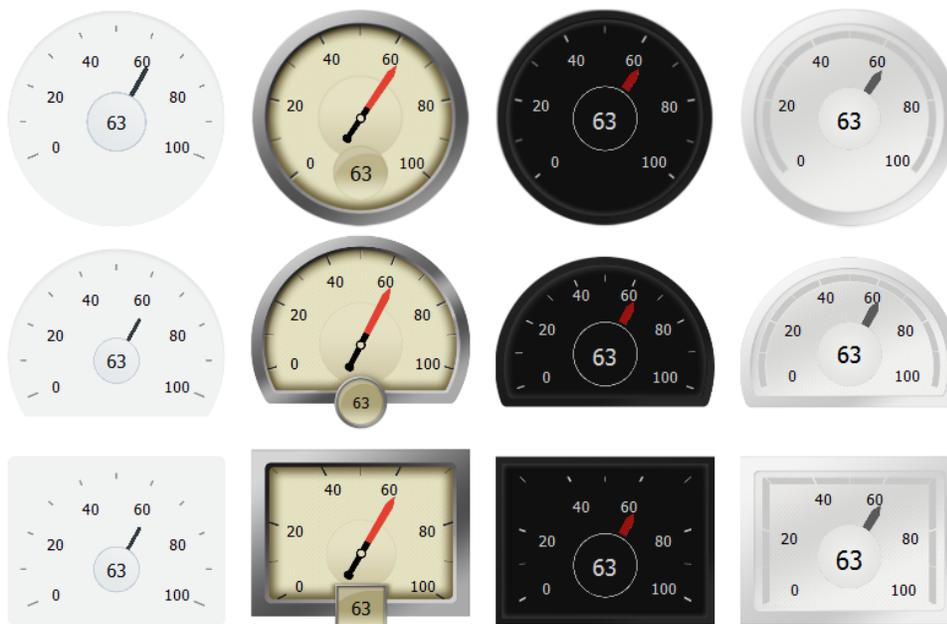
The gauge components also support interactivity features such as popup and context menus. All gauges also include value change support, which allows the user to change the gauge's metric value.

Gauge Shape Variations

Gauge shapes are configurable and vary by gauge type.

- Dial gauges: The `background` attribute sets shape and shading. [Figure 25-5](#) shows the backgrounds available for dial gauges, which include the circle, dome, and rectangular shapes with alta, antique, dark, or light shading.

Figure 25-5 Dial Gauge Backgrounds



- LED gauge: The `type` attribute sets the LED gauge's shape. [Figure 25-6](#) shows the types available for the LED gauge, which include circle, rectangle, arrow, triangle, diamond, and star.

Figure 25-6 LED Gauge Types



- Status meter gauge: The `orientation` attribute determines whether the status meter is displayed along a horizontal, a circular, or a vertical axis. [Figure 25-4](#) shows status meter gauges configured as horizontal bars, circles and vertical bars.
- Rating gauge: The rating gauge's `shape` attribute determines the rating gauge's shape. [Figure 25-3](#) shows the shapes available for the rating gauge, which include the star, diamond, circle, rectangle, and triangle shapes.

Gauge Thresholds

Gauge thresholds contain values that represent the upper bound of a range on the status meter or LED gauge. Typically, thresholds are defined to indicate whether a gauge's metric value falls within an acceptable range.

Figure 25-7 shows three each of horizontal, circular and vertical status meter gauges, each configured to show a red indicator when the gauge's metric value is at or below 30, a yellow indicator when the metric value is at or below 60, and a green indicator when the metric value is greater than 60.

Figure 25-7 Status Meter Gauge Thresholds

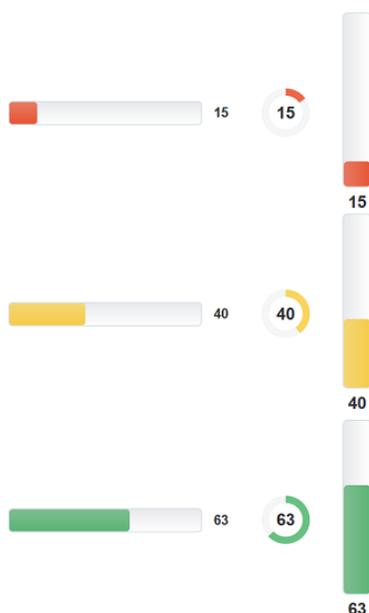


Figure 25-8 shows three LED gauges, each configured to display its shape in red when the gauge's metric value is at or below 30, in yellow when the metric value is at or below 60, and in green when the metric value is greater than 60.

Figure 25-8 LED Gauge Thresholds



Gauge Visual Effects

By default, gauges apply gradients and overlays to color display. You can disable visual effects to achieve a flatter design.

The second row in [Figure 25-9](#) shows the result of disabling visual effects on the three LED gauges in [Figure 25-8](#).

Figure 25-9 LED Gauges Showing Enabled and Disabled Visual Effects



Active Data Support (ADS)

Gauges support ADS by sending a Partial Page Refresh (PPR) request when an active data event is received. The PPR response updates the components, animating the changes as needed. Supported ADS events include updates to the `value`, `minimum`, and `maximum` gauge attributes.



Note:

EL Operators are not supported within the EL Expressions for active attributes.

For additional information about using the Active Data Service, see [Using the Active Data Service with an Asynchronous Backend](#).

Gauge Animation

Gauges support animation on initial display or data change using the `af:transition` child tag.

Gauge Tooltips

You can configure a tooltip of contextual information to display when a user moves a cursor over the gauge. This is done using the `shortDesc` attribute.

[Figure 25-10](#) shows the tooltip for a dial gauge.

Figure 25-10 Circular Status Meter Gauge Displaying a Tooltip



Gauge Popups and Context Menus

You can configure gauges to display popups or context menus using the `af:showPopupBehavior` tag.

Figure 25-11 shows a circular status meter gauge configured to show a popup when the user clicks the gauge. The popup displays an output message in a note window.

Figure 25-11 Status Meter Gauge Displaying a Popup

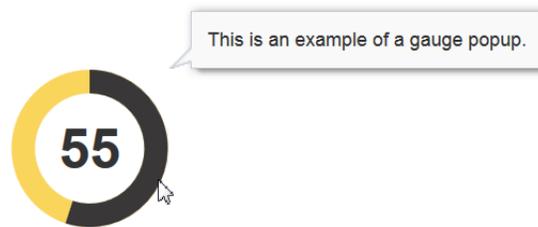
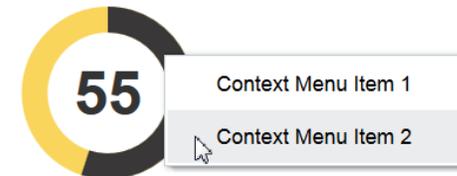


Figure 25-12 shows a circular status meter gauge configured to show a context menu when the user right-clicks the gauge. The context menu displays an output message in a note window.

Figure 25-12 Status Meter Gauge Displaying a Context Menu



Gauge Value Change Support

You can configure gauges to accept input from the user to change the metric value. For example, you could configure a rating gauge that allows the user to assign the number of stars to a movie rating.

Figure 25-13 shows an example of a rating gauge configured to accept user input. The gauge's initial metric value is 1. As the user glides the mouse over the stars, the color changes to indicate the number of selected stars. To effect the change, the user clicks the highlighted star with the largest desired value.

Figure 25-13 Rating Gauge Illustrating Value Change Support



Gauge Reference Lines (Status Meter Gauges)

You can configure status meter gauges to display a reference line. Reference lines can be used in conjunction with thresholds to display trend information, such as the previous value, and target information at the same time.

Figure 25-14 shows a horizontal, a circular, and a vertical status meter gauge configured to show reference lines. Each gauge is configured to show one black reference line with its value set to 65.

Figure 25-14 Status Meter Gauges Displaying Reference Lines



Additional Functionality of Gauge Components

You may find it helpful to understand other ADF Faces features before you implement your gauge component. Additionally, once you have added a gauge component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that gauge components can use:

- Partial page rendering: You may want a gauge to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: When enabled for users to change the way the gauge displays at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: You can make your gauge components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Skins and styles: You can customize the appearance of gauge components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound gauges based on how your ADF Business Components are configured. For more information, see "Creating Databound

Gauges" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see "Designing a Page Using Placeholder Data Controls" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Gauge Component

To use the ADF DVT Gauge component, add the gauge to a page using the Component Palette window. Then define the data for the gauge and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Gauge Component Data Requirements

Gauges display the following kinds of data values:

- **Metric:** The value that the gauge is to plot. This value can be specified as static data in the `Value` attribute in the Properties window. It can also be specified through data controls. This is the only required data for a gauge.
- **Minimum and maximum:** Optional values that identify the lowest and highest points on the gauge value axis. These values can be provided as dynamic data from a data collection. They can also be specified as static data in the `Minimum` and `Maximum` fields in the Properties window for the gauge.

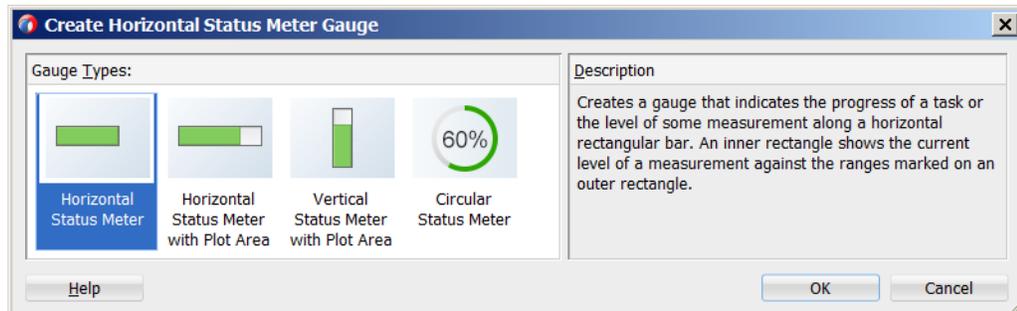
The default minimum value for all gauges is 0. The default maximum value for dial, LED, and status meters gauges is 100, and the default maximum value for rating gauges is 5. You must change this value if your gauge's metric exceeds the maximum value for your gauge to display properly.

- **Thresholds:** Optional values that can be provided as dynamic data from a data collection to identify ranges of acceptability on the value axis of the gauge. You can also specify these values as static data using gauge threshold tags in the Properties window. For more information, see [How to Configure Gauge Thresholds](#).

How to Add a Gauge to a Page

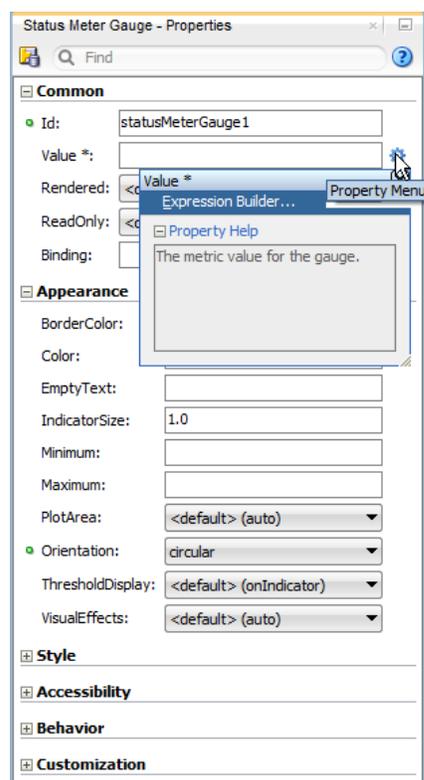
When you are designing your page using simple UI-first development, you use the Components window to add a gauge to a JSF page. When you drag and drop a gauge component onto the page, a Create Gauge dialog displays available categories of gauge types, with descriptions, to provide visual assistance when creating gauges. [Figure 25-15](#) shows the Create Horizontal Status Meter Gauge dialog.

Figure 25-15 Create Gauge Dialog for Horizontal Status Meter



Once you complete the dialog, and the gauge is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the gauge.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. [Figure 25-16](#) shows the Property menu for a gauge component `value` attribute.

Figure 25-16 Gauge Value Attribute in Properties Window **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a gauge and the binding will be done for you. For more information, see "Creating Databound Gauges" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child components can affect functionality. For more information, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a gauge to a page:

1. In the ADF Data Visualizations page of the Components window, from the Gauge panel, drag and drop the desired gauge onto the page to open the Create Gauge dialog.
2. In the Create Gauge dialog, click **OK** to add the gauge to the page.
3. In the Properties window, view the attributes for the gauge. Use the **Component Help** button to display the complete tag documentation for the `gauge` component.

What Happens When You Add a Gauge to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a gauge from the Components window onto a JSF page.

For all gauges but the rating gauge, JDeveloper adds the `dvt:gaugeMetricLabel` tag to the page. For the status meter and LED gauges which support thresholds, JDeveloper adds one `dvt:gaugeThreshold` tag representing the maximum bound of the gauge.

The example below shows the default code that JDeveloper adds to the page for a dial, LED, status meter, and rating gauge. In this example, each gauge is contained in a grid cell of the `af:panelGridLayout` component. The gauge-related tags are highlighted in bold.

```
<af:panelGridLayout id="pgl1">
  <af:gridRow marginTop="5px" height="auto" id="gr1">
    <af:gridCell marginStart="5px" width="50%" id="gc1">
      <b>dvt:dialGauge id="dialGauge1"</b>
      <b>dvt:gaugeMetricLabel rendered="true" id="gml1"/>
    </af:gridCell>
    <af:gridCell marginStart="5px" width="50%" id="gc2">
      <b>dvt:ledGauge id="ledGauge1" type="circle"</b>
      <b>dvt:gaugeThreshold id="thr1" color="#d62800"/>
      <b>dvt:gaugeMetricLabel rendered="true" id="gml2"/>
    </af:gridCell>
  </af:gridRow>
  <af:gridRow marginTop="5px" height="auto" marginBottom="5px" id="gr2">
    <af:gridCell marginStart="5px" width="50%" id="gc4">
      <b>dvt:statusMeterGauge id="statusMeterGauge1"</b>
      <b>dvt:gaugeThreshold id="thr3" color="#d62800"/>
      <b>dvt:gaugeMetricLabel rendered="true" id="gml3"/>
    </af:gridCell>
    <af:gridCell marginStart="5px" width="50%" marginEnd="5px" id="gc6">
      <b>dvt:ratingGauge id="ratingGauge1"/>
    </af:gridCell>
  </af:gridRow>
</af:panelGridLayout>
```

For information about the panel grid layout component, see [How to Use the panelGridLayout, gridRow, and gridCell Components to Create a Grid-Based Layout](#).

How to Add Data to Gauges

A gauge's metric value is the only required value for a gauge, and you specify this value in the gauge's `value` attribute. You can specify the value for the `value` attribute

as a static numeric value in the Properties window, in a managed bean that returns the gauge's numeric value, or by binding a data control to a gauge.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To add data to a gauge:

1. Optionally, create the managed bean that will return the gauge's metric value.
If you need help creating classes, see "Working with Java Code" in *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
2. In the Structure window, right-click the **dvt:typeGauge** node and choose **Go To Properties**.
3. In the Properties window, do one of the following:
 - To add data to the gauge statically or to reference a managed bean that returns the gauge's metric value, expand the **Common** section and specify the metric value in the **Value** field.

You can enter a static numeric value or specify an EL expression that references the managed bean and metric value.

For example, to specify an EL expression for a managed bean named `gauge` that returns the gauge's metric value in a method named `getGaugeMetric()`, enter the following in the **Value** field: `{gauge.gaugeMetric}`.

For help with creating EL expressions, see [How to Create an EL Expression](#).
 - To bind the gauge to a data control, click **Bind to ADF Control** to select a data collection.

For more information about using data controls to supply data to your gauge, see "Creating Databound Gauges" in *Developing Fusion Web Applications with Oracle Application Development Framework*.
4. If needed, in the Properties window, expand the **Appearance** section and enter values for the **Minimum** and **Maximum** fields.

You can enter a static numeric value or specify an EL expression that references a managed bean and minimum or maximum value.

 **Note:**

If your gauge's metric value exceeds the default maximum value for the gauge, you must change the maximum value or your gauge will not display properly.

If you want your LED or status meter gauge to display thresholds for predefined ranges, add a `dvt:gaugeThreshold` tag for each threshold. For additional information, see [How to Configure Gauge Thresholds](#).

Configuring Gauges

The properties for the gauge component are sufficient to produce a gauge, but you can modify the gauge properties to customize the display and behavior of the gauge. You can also add and configure child components or supported facets to customize the display and behavior of the gauge. The prefix `dvt:` occurs at the beginning of each gauge component name indicating that the component belongs to the *Tag Reference for Oracle ADF Faces Data Visualization Tools* tag library.

Configurable elements vary by gauge type. Some elements are available to all or more than one gauge type and include:

- `minimum` and `maximum`: Attributes that specify the minimum and maximum values for the gauge.
- `inlineStyle`: Attribute that specifies the style of the outer element (enclosing `div`) of the component.
- `shortDesc`: Attribute that specifies the short description of this component. This is used to customize the tooltip text that appears when the user hovers the mouse over the gauge.
- `visualEffects`: Attribute that specifies whether gradients and overlays are displayed. By default, this is set to `auto`, but you can also set this to `none` to turn off visual effects.
- `styleClass`: Attribute that sets a CSS style class to use for this component.
- `gaugeMetricLabel`: Child component that determines the metric label's style, visibility, scaling, and text type. This attribute is not available on the rating gauge which uses the number of shapes as its metric value.
- `readOnly`: Attribute on dial, rating, and status meter gauges that determines whether the user can change the gauge's metric value.
- `gaugeThreshold`: Child component that specifies a threshold for LED or status meter gauges.
- `borderColor`: Attribute that specifies the border color of the LED or status meter gauge's indicator.
- `color`: Attribute that specifies the fill color of the LED or status meter gauge's indicator.
- `orientation`: Attribute that determines the shape direction of rating or status meter gauges. This specifies whether rating gauges are rendered vertically or horizontally, and whether status meter gauges are rendered vertically, horizontally, or in a circular manner.
- `title`: Attribute that accepts a string value and defines the title for LED and status meter gauges.
- `titleStyle`: Attribute that determines the styling options on the string value set in the `title` attribute.

You can view a complete list of gauge tags and supported child components by clicking **Component Help** in the Properties window for the gauge.

Configuring Dial Gauges

Configurable elements specific to dial gauges include:

- `background`: Attribute that determines the shape and background style of the dial gauge.
- `indicator`: Attribute that specifies the indicator style for the dial gauge.
- `gaugeTickLabel`: Child component that determines the tick label's style, visibility, scaling, and text type.

Configuring LED Gauges

Configurable elements specific to LED gauges include:

- `type`: Attribute that specifies the shape of the LED gauge.
- `size`: Attribute that determines the relative size of the LED gauge.
- `rotation`: Attribute that specifies the rotation of LED gauges configured to use arrow or triangle shapes.

Configuring Rating Gauges

Configurable elements specific to rating gauges include:

- `shape`: Attribute that determines the shape of the rating gauge increments.
- `inputIncrement`: Attribute that specifies the change increment when users edit the metric value.
- `unselectedShape`: Attribute that determines the shape of gauge increments that are not selected.
- `changedStyle`: Attribute that determines the color and border color of selected gauge increments after a change in value has been made.
- `selectedStyle`: Attribute that determines the color and border color of gauge increments that have been selected.
- `unselectedStyle`: Attribute that determines the color and border color of gauge increments that are not selected.
- `hoverStyle`: Attribute that controls the color and border color of gauge increments that are hovered on but not yet selected.

Configuring Status Meter Gauges

Configurable elements specific to status meter gauges include:

- `indicatorSize`: Attribute that determines the size of the indicator, relative to the plot area. Accepts a positive numeric value.
- `borderRadius`: Attribute that determines the degree of curvature of the corners of the indicator. Accepts a positive pixel value.
- `plotArea`: Attribute that determines whether or not the plot area is displayed.

- `plotAreaBorderRadius`: Attribute that determines the degree of curvature of the corners of the plot area. Accepts a positive pixel value. If this value isn't specified, the `borderRadius` attribute applies to both the indicator and the plot area.
- `plotAreaColor`: Attribute that determines the color of the plot area. Accepts a color value or hex code.
- `plotAreaBorderColor`: Attribute that determines the color of the plot area border. Accepts a color value or hex code.
- `thresholdDisplay`: Attribute that determines how thresholds are displayed.
- `titlePosition`: Attribute that controls the placement of the string value defined in the `title` attribute. Accepts the values `auto`, `center`, and `start`.

There are configurable elements specific to circular status meter gauges which are ignored by other status meter gauges. These include:

- `innerRadius`: Attribute that determines the radius of an inner circle in the gauge. Accepts a value between 0 and 1.
- `startAngle`: Attribute that determines the angle at which the indicator starts being drawn in a clockwise manner (and counterclockwise for BiDi locales). Accepts a value between 0 and 360.
- `angleExtent`: Attribute that determines the extent to which the plot area is drawn. The plot area is drawn starting at the `startAngle` and ending after the full angle extent is drawn. Accepts a value between 0 and 360.

Customizing Gauge Display Elements

ADF DVT Gauges provide a number of useful customizations for display elements, including thresholds, styles, numeric data values, visual effects, animation, and reference lines.

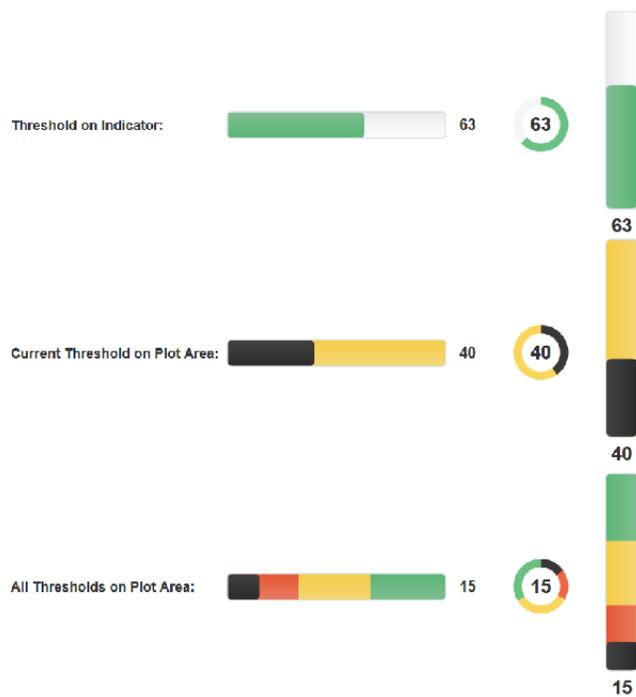
How to Configure Gauge Thresholds

Thresholds are numerical data values in status meter or LED gauges that highlight a particular range of values. Thresholds must be values between the minimum and the maximum value for a gauge.

On status meter gauges, the range identified by a threshold is filled with a color that is different than the color of other ranges. The threshold can be displayed on the indicator or the plot area.

[Figure 25-17](#) shows horizontal, circular, and vertical status meters configured for thresholds. The gauges in the first row are configured to show the threshold on the indicator. In the second row, the gauges are configured to display the current threshold on the plot area. In the third row, the gauges are configured to display all thresholds on the plot area.

Figure 25-17 Status Meter Gauge Threshold Display Options



On LED gauges, the background is filled with the color defined for the threshold range containing the metric value. [Figure 25-18](#) shows three LED gauges configured with thresholds using the same maximum bound and color values as the status meters shown in [Figure 25-17](#).

Figure 25-18 LED Gauges Configured with Three Thresholds



When you create a status meter or LED gauge by dragging the gauge to the page from the Components window, JDeveloper adds one `dvt:gaugethreshold` tag as a child of the gauge. You can add additional thresholds by inserting additional `dvt:gaugethreshold` tags and defining the threshold's `value` and `color` attributes.

The data collection for a gauge can provide dynamic values for thresholds when the gauge is databound. For information about using dynamic values for thresholds, see "Creating Databound Gauges" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To configure status meter or LED gauge thresholds:

1. In the Structure window, right-click the **dvt:statusMeterGauge** or **dvt:LEDGauge** component and choose **Insert Inside (Status Meter or LED) Gauge > Threshold**.

Depending upon how you created the gauge, you may already have one `dvt:gaugeThreshold` component defined and can proceed to the next step for that threshold.

2. Right-click the **dvt:gaugeThreshold** node and choose **Go to Properties**.
3. In the Properties window, set values for the following:

- **Maximum:** Specify the maximum bound for the threshold. You can enter an integer value or use the dropdown menu to choose **Expression Builder** to enter an EL Expression that represents the maximum bound.

You do not need to enter a value for the threshold representing the maximum bound for the gauge. This value is automatically derived from the value in the gauge's `maximum` attribute.

- **Color:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.

For example, enter `#0000FF` to render the threshold in blue.

- **BorderColor:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.

For example, enter `#000000` to render the threshold's border color in black.

- **ShortDesc:** Specify the custom hover text for the threshold in plain text or choose **Expression Builder** from the attribute's dropdown menu to enter an EL expression that contains the text value.

4. Repeat Step 1 through Step 3 for each threshold that you want to configure.
5. For status meter gauges, set the following attributes to customize the threshold display.

- a. In the Structure window, right-click the **dvt:statusMeterGauge** and choose **Go to Properties**.

- b. In the Properties window, in the **ThresholdDisplay** field, use the dropdown menu to customize the placement and appearance of the threshold.

By default, the threshold is displayed on the indicator. If you want the current threshold displayed on the plot area, choose `currentOnly`. To display all thresholds on the plot area, choose `all`.

- c. In the **PlotArea** field, use the dropdown menu to turn the plot area display on or off.

By default, **PlotArea** is set to `auto` which will show the plot area when thresholds are configured and **ThresholdDisplay** is set to `currentOnly` or `all`. You can also set this to `on` or `off`.

- d. In the **Indicator** field, enter any positive value to change the relative size of the indicator.

For example, to set the indicator to consume 50% of the plot area, enter `0.5`, or to make the indicator twice as large as the plot area, enter `2`. [Figure 25-19](#) shows the effect of changing the indicator size to `0.5`.

Figure 25-19 Status Meter Gauge Indicator Size Set to 0.5



Formatting Gauge Style Elements

You can customize the styling of gauges to change the initial size of a gauge and apply style elements to labels and other presentation features.

How to Change Gauge Size and Apply CSS Styles

You can customize the width and height of a gauge by applying a CSS style or specifying a value in the gauge's `inlineStyle` attribute.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To specify the size of or apply CSS styles to a gauge:

1. In the Structure window, right-click the gauge component and choose **Go to Properties**.
2. To specify size: In the Properties window, expand the **Style** section. Specify the initial size of the gauge in the **InlineStyle** attribute. You can specify a fixed size or specify a relative percent for both width and height.

For example, to create a gauge that fills 50% of its container's width and has a height of 200 pixels, use the following setting for the **InlineStyle** attribute:

```
width:50%;height:200px
```

 **Best Practice Tip:**

Instead of specifying width at 100% in the `inlineStyle` attribute, set the `styleClass` attribute to `AFStretchWidth`.

3. To apply CSS styles: In the Properties window, expand the **Style** section and enter the name of the style class in the **StyleClass** field.

For example, to set the width of the gauge to fill 100% of its container's width, use the following setting for the **StyleClass** attribute:

```
AFStretchWidth
```

For information about applying CSS styles, see [Customizing the Appearance Using Styles and Skins](#).

How to Format Gauge Text

You can format the text in the following gauge child components:

- `gaugeMetricLabel`
- `gaugeTickLabel`

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To format gauge text:

1. In the Structure window, right-click **dvt:gaugeMetricLabel** or **dvt:gaugeTickLabel** and choose **Go to Properties**.
2. In the Properties window, enter a value for **LabelStyle**.

This property accepts font-related CSS attributes such as `font-weight` and `font-size`. For example, to set the label to bold, enter the following for **LabelStyle**:

```
font-weight:bold;
```

3. For status meter gauges, in the Properties window, select a value for **Position**.

This property controls the position of the label on the gauge. Possible options are: `auto`, `center`, `insideIndicatorEdge`, `outsideIndicatorEdge`, `outsidePlotArea`, and `withTitle`. `auto` will default to `outsidePlotArea` for horizontal/vertical and `center` for circular status meter gauges.

You can also set the style attributes of gauge components globally across all pages in your application by using a cascading style sheet (CSS) to build a skin, and configuring your application to use the skin. By applying a skin to define the styles used in gauge components, the pages in an application will be smaller and more

organized, with a consistent style easily modified by changing the CSS file. For more information, see [What You May Need to Know About Skinning and Formatting Gauge Style Elements](#).

What You May Need to Know About Skinning and Formatting Gauge Style Elements

You can set the font and other style attributes of gauge components globally across all pages in your application by using a cascading style sheet (CSS) to build a skin and configuring your application to use the skin. By applying a skin to define the styles used in gauge components, the pages in an application will be smaller and more organized, with a consistent style easily modified by changing the CSS file.

You can use the ADF Data Visualization Tools Skin Selectors to define the styles for gauge components. Gauge component skin selectors that support styling include the following:

- `af|dvt-dialGauge`
- `af|dvt-ledGauge`
- `af|dvt-ratingGauge`
- `af|dvt-ratingGauge::selected-shape`
- `af|dvt-ratingGauge::unselected-shape`
- `af|dvt-ratingGauge::hover-shape`
- `af|dvt-ratingGauge::changed-shape`
- `af|dvt-statusMeterGauge`
- `af|dvt-gaugeMetricLabel`
- `af|dvt-gaugeTickLabel`
- `af|dvt-gaugeThreshold`
- `af|dvt-gaugeThreshold::index$`

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To use a custom skin to set gauge styles:

1. Add a custom skin to your application containing the defined skin style selectors for the gauge subcomponents.

For example, specify the font family for all gauge metric labels in a `mySkin.css` file as follows:

```
af|dvt-gaugeMetricLabel
{
  font-weight:bold;
}
```

For help with creating a custom skin, see the "Creating an ADF Skin File" section in *Developing ADF Skins*.

2. Configure the application to use the custom skin in the `trinidad-config.xml` file.

For help with configuring the application, see the "Applying the Finished ADF Skin to Your Web Application" chapter of *Developing ADF Skins*.

For additional information about using styles and skins, see [Customizing the Appearance Using Styles and Skins](#).

How to Format Numeric Data Values in Gauges

You can use the gauge's `gaugeMetricLabel` child component to format the appearance of the gauge's metric value in dial, LED, and status meter gauges. You can also format the appearance of dial gauge tick labels using the `gaugeTickLabel` child tag. Each component has a `textType` attribute that lets you specify whether you want to display the metric value itself or a percentage that the value represents.

In some cases, this might be sufficient numeric formatting, but you can use properties on the component to change scaling.

If you wish to further format the gauge metric or tick label value, you can use an ADF Faces standard converter, `af:convertNumber`. For example, you may wish to display the value as currency or display specific decimal settings.

You may also specify a custom label using the `text` attribute, in which case the `textType` attribute will be ignored.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a dial, LED, or status meter gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To format numeric values in a gauge:

1. If you are formatting a dial gauge's tick label, in the Structure window, right-click the **dvt:dialGauge** component and choose **Insert Inside Dial Gauge > Gauge Tick Label**.
2. In the Structure window, right-click the **dvt:gaugeMetricLabel** or **dvt:gaugeTickLabel** node and choose **Go to Properties**.
3. In the Properties window, set values for the following:
 - **Scaling**: Use the attribute's dropdown list to change the default scaling from auto. You can select one of the available scaling options or `none` to turn off scaling.

- **TextType**: Use the attribute's dropdown list to change the default text type from `number` to `percent`.
4. If you want to specify additional formatting for the data values displayed in the gauge metric or tick label, do the following:
 - a. In the Structure window, right-click the **dvt:gaugeMetricLabel** or **dvt:gaugeTickLabel** node and choose **Insert Inside (Gauge Metric Label or Gauge Tick Label) > Convert Number**.
 - b. Right-click the **af:convertNumber** node and choose **Go to Properties**.
 - c. In the Properties window, specify values for the attributes of the **af:convertNumber** component to produce additional formatting. Click **Help** or press F1 to display the complete tag documentation for the `af:convertNumber` component.
 5. Alternatively, you can specify a custom label. In the Properties window of the **dvt:gaugeMetricLabel** node, specify a value for the **Text** attribute. If the **Text** attribute is specified, the **TextType** attribute will be ignored.

How to Disable Gauge Visual Effects

By default, gauges are displayed with gradient coloring and overlays as shown in [Figure 25-9](#). To disable these visual effects, set the gauge's `visualEffects` attribute to `none`.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To disable visual effects on a gauge:

1. In the Structure window, right-click the **dvt:typeGauge** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section, and from the **VisualEffects** attribute's dropdown list, select `none`.

How to Configure Gauge Animation

To configure gauge animation, add the `af:transition` tag as a child of the gauge component and configure the trigger type and transition effect.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To configure gauge animation:

1. In the Structure window, click the gauge component.
2. In the Source editor, add the `af:transition` tag as a child of the highlighted gauge component as shown in the following example.

```
<af:transition triggerType="display" transition="auto"/>
```

How to Configure Status Meter Gauge Reference Lines

You can add reference lines to status meter gauges at specified values on the gauge's axis. To configure reference lines, add the `dvt:referenceLine` component to the status meter gauge and configure as needed.

The example below shows the code on the JSF page that creates the gauge and reference lines in [Figure 25-14](#).

```
<af:panelGroupLayout id="pg11" layout="horizontal">
  <af:spacer width="5" id="s1"/>
  <dvt:statusMeterGauge value="90" indicatorSize="0.5" plotArea="on" id="smg1">
    <dvt:referenceLine color="#FFFFFF" value="75" id="r11"/>
    <dvt:referenceLine color="#000000" value="95" id="r12"/>
    <dvt:gaugeThreshold maximum="33" id="gt1"/>
    <dvt:gaugeThreshold maximum="67" id="gt2"/>
    <dvt:gaugeThreshold id="gt3"/>
  </dvt:statusMeterGauge>
  <af:spacer width="25" id="s2"/>
  <dvt:statusMeterGauge inlineStyle="width:50px;height:50px;"
    orientation="circular" value="90" id="smg2">
    <dvt:referenceLine color="#FFFFFF" value="75" id="r13"/>
    <dvt:referenceLine color="#000000" value="95" id="r14"/>
    <dvt:gaugeThreshold maximum="33" id="gt4"/>
    <dvt:gaugeThreshold maximum="67" id="gt5"/>
    <dvt:gaugeThreshold id="gt6"/>
  </dvt:statusMeterGauge>
</af:panelGroupLayout>
```

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To add a reference line to a status meter gauge:

1. In the Structure window, right-click the **dvt:statusMeterGauge** and choose **Insert Inside Status Meter Gauge > Reference Line**.
2. Right-click the **dvt:referenceLine** node and choose **Go to Properties**.
3. In the Properties window, enter values for the following:
 - **Value:** Specify the value for the reference line or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the value.
 - **Color:** Specify the RGB value in hexadecimal notation or choose **Expression Builder** from the attribute's dropdown menu to enter an expression that evaluates to the RGB value.
For example, to render the reference line in white, enter #000000.
 - **LineStyle:** Use the attribute's dropdown list to change the line from `solid` to `dashed` or `dotted`.
 - **LineWidth:** Specify the width of the line.

Adding Interactivity to Gauges

ADF DVT Gauges can be made more interactive and user-friendly by adding tooltips, popups, context menus, and value change support.

How to Configure Gauge Tooltips

You configure a gauge tooltip by setting a value for the gauge's `shortDesc` attribute. You may also configure a custom tooltip for a gauge threshold by setting a value for the threshold's `shortDesc` attribute. Note that the threshold tooltip will override the parent gauge's tooltip.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

To add a tooltip to a gauge and gauge threshold:

1. In the Structure window, right-click the **dvt:typeGauge** component and choose **Go to Properties**.
2. In the Properties window, expand the **Accessibility** section.
3. In the **ShortDesc** field, enter a description for the gauge. You can also use the attribute's dropdown menu to open the Select Text Resource or Expression Builder dialogs to select a text resource or EL expression that contains the gauge's description.
4. In the Structure window, right-click the child **dvt:gaugeThreshold** component and choose **Go to Properties**.

5. In the Properties window, expand the **Other** section.
6. In the **ShortDesc** field, enter a description for the gauge threshold. You can also use the attribute's dropdown menu to open the Expression Builder dialog to select an EL expression that contains the gauge's description.

How to Add a Popup or Context Menu to a Gauge

The process to add a popup or context menu is essentially the same. Add the `af:showPopupBehavior` tag as a child of the gauge component, define the trigger type as `click` for popup menus or `contextMenu` for context menus, and add an `af:popup` containing the desired behavior to the page.

The example below shows the code on the page for the popup menu shown in [Figure 25-11](#). In this example, the `af:showPopupBehavior` component uses the `popupId` to reference the `af:popup` component. The `af:popup` component is configured with the `af:noteWindow` component which is configured to display a simple message in the `af:outputFormatted` component. The `triggerType` of the `af:showPopupBehavior` tag is set to `click`, and the note window will launch when the user clicks anywhere in the gauge.

```
<af:group id="g1">
  <dvt:dialGauge id="dialGauge1" value="63" shortDesc="Dial Gauge with Popup">
    <af:showPopupBehavior popupId="::noteWindowPopup"
      triggerType="click"/>
    <dvt:gaugeMetricLabel rendered="true" id="gml1"/>
  </dvt:dialGauge>
  <af:popup childCreation="deferred" autoCancel="disabled"
    id="noteWindowPopup" launcherVar="source"
    clientComponent="true" eventContext="launcher">
    <af:noteWindow id="nw1">
      <af:outputFormatted value="This is an example of a gauge popup."
        id="of1"
          shortDesc="Gauge Popup Example"/>
    </af:noteWindow>
  </af:popup>
</af:group>
```

You can change the popup to the context menu displayed in [Figure 25-12](#) by simply changing the trigger type for the `af:showPopupBehavior` component to `contextMenu` as shown in the following code snippet:

```
<af:showPopupBehavior popupId="::noteWindowPopup" triggerType="contextMenu"/>
```

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

Add a popup component to your page. For help with configuring the `af:popup` component, see [Using Popup Dialogs, Menus, and Windows](#).

To add a popup or context menu to a gauge:

1. In the Structure window, right-click the gauge component and choose **Insert Inside Type Gauge > Show Popup Behavior**.
2. Right-click `af:showPopupBehavior` and choose **Go to Properties**.
3. In the Properties window, enter values for the following:
 - **PopupId**: Specify the ID of the `af:popup` component.
 - **TriggerType**: For popup menus, enter `click`. For context menus, enter `contextMenu`.

Optionally, set values for **Align**, **AlignId**, and **Disabled**. Click **Component Help** for more information about the `af:showPopupBehavior` component.

How to Configure Value Change Support for a Gauge

You can allow users to change the metric value for a dial or rating gauge by setting the gauge's `readOnly` attribute to `false`.

To process the change on the server, specify a listener in the gauge's `valueChangeListener` attribute.

The code below shows an example of a value change listener for a rating gauge configured to process a value change on the server. In this example, the rating gauge and listener are contained in a managed bean named `gaugeData`. The listener is named `ratingChangedListener` and simply outputs the new value to the console.

```
import javax.faces.event.ValueChangeEvent;

public class GaugeData {
    private Double gaugeValue = 3.0;
    public void ratingChangedListener (ValueChangeEvent e){
        if (e != null){
            gaugeValue = (Double) e.getNewValue();
            System.out.println("You clicked on " + gaugeValue + " stars");
        }
    }
    public Double getGaugeValue(){
        return gaugeValue;
    }
}
```

The example below shows the code on the JSF page for a rating gauge configured to use the `gaugeData` managed bean.

```
dvt:ratingGauge id="ratingGauge1" readOnly="false" value="#{gaugeData.gaugeValue}"
    valueChangeListener="#{gaugeData.ratingChangedListener}">
</dvt:ratingGauge>
```

You can also process the change on the client using `af:clientListener` components configured for `valueChange` and `input` event types. To use this method, you must create JavaScript functions that perform change event handling.

The code below shows an example of a rating gauge configured with two `af:clientListener` components, one to handle the `valueChange` event and one to

handle the `input` event. In this example, the `valueChangeListener` function sends an alert to the browser, and the `inputListener` function sends a message to the browser's console log.

```
<script type="text/javascript" xmlns="http://www.w3.org/1999/xhtml">
  function valueChangeListener(event) {
    alert("valueChange for " + event.getSource().getId() + ": " +
      event.getOldValue() + " --> " + event.getNewValue());
  }
  function inputListener(event) {
    console.log("input for " + event.getSource().getId() + ": " +
      event.getValue());
  }
</script>
<dvt:ratingGauge id="ratingGauge1" readOnly="false" value="#{gauge.gaugeValue}">
  <af:clientListener method="valueChangeListener" type="valueChange"/>
  <af:clientListener method="inputListener" type="input"/>
</dvt:ratingGauge>
```

 **Note:**

This example uses inline JavaScript for the purposes of illustration only. Inline JavaScript can increase response payload size, will never be cached in the browser, and can block browser rendering. Instead of using inline JavaScript, consider putting all scripts in JavaScript libraries. For additional information about adding JavaScript to a page, see [Adding JavaScript to a Page](#). For additional information about client-side event handling, see [Listening for Client Events](#).

Before you begin:

It may be helpful to have an understanding of how gauge attributes and child tags can affect functionality. For more information about configuring gauges, see [Configuring Gauges](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality of Gauge Components](#).

Add a gauge to your page. For help with adding a gauge to a page, see [How to Add a Gauge to a Page](#).

If you are configuring server-side event handling, add the value change listener to the gauge's managed bean. If you need help with managed beans, see [Creating and Using Managed Beans](#).

If you are configuring client-side event handling, create the JavaScript functions that will handle the `valueChange` and `input` events. For additional information about client-side event handling, see [Listening for Client Events](#).

To configure value change support for a gauge:

1. If you are configuring server-side change support, do the following:
 - a. In the Structure window, right-click the gauge component and choose **Go to Properties**.

- b. In the Properties window, expand the **Common** section if needed.
- c. From the **ReadOnly** attribute's dropdown list, select `False`.
- d. Expand the **Behavior** section.
- e. From the **ValueChangeListener** attribute's dropdown menu, choose **Edit** to select the gauge's managed bean and listener.

For example, to reference the `gaugeData` sample bean, select `gaugeData` from the **Managed Bean** field's dropdown list, and select `ratingChangedListener` for the method. You can also choose **Expression Builder** from the **ValueChangeListener** attribute's dropdown menu to enter an EL expression that evaluates to the gauge's listener.

2. If you are configuring client-side event support, do the following:
 - a. In the Structure window, right-click the gauge component and choose **Insert Inside Type Gauge > Client Listener**.
 - b. In the Insert Client Listener dialog, enter the name of the JavaScript function that will handle the event and the event type.

For example, to use the example `valueChangeListener()` function as shown in the code sample above, enter `valueChangeListener` for the method and `valueChange` for the event type.
 - c. Repeat Step 1 and Step 2 to insert additional client listeners as needed.
 - d. In the Structure window, right-click the gauge component and choose **Go to Properties**.
 - e. In the Properties window, expand the **Common** section if needed.
 - f. From the **ReadOnly** attribute's dropdown list, select `False`.

Using NBox Components

This chapter describes how to use the ADF Data Visualization NBox component to display data in NBoxes using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of these components.

If your application uses the Fusion technology stack, you can use data controls to create NBoxes. For more information, see "Creating Databound NBox Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About ADF Data Visualization NBox Components](#)
- [Using the NBox Component](#)

About the NBox Component

The ADF DVT NBox Component is a data grouping visualization that utilizes two ranges of data to form a grid of cells, and each cell contains customizable nodes that represent individual data items. NBox components are useful to group data with multiple factors of consideration, such as employee potential against employee performance.

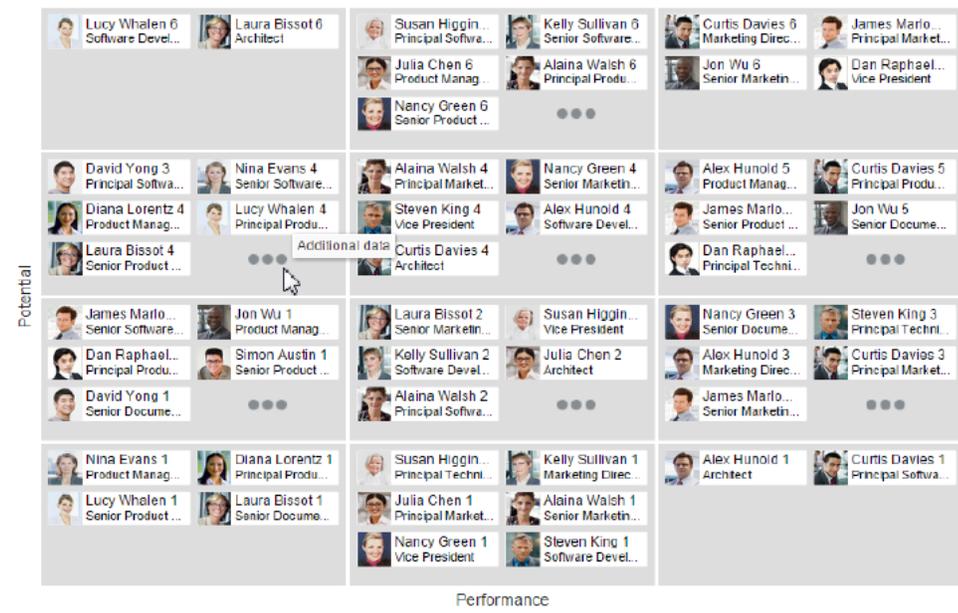
ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

NBox Use Cases and Examples

The `nBox` component is comprised of two parts: the node that represents the data and the grid that comprises the cells into which the nodes are placed. If the number of nodes is greater than the space allocated for the cell, the NBox displays an indicator that users can click to access the additional nodes.

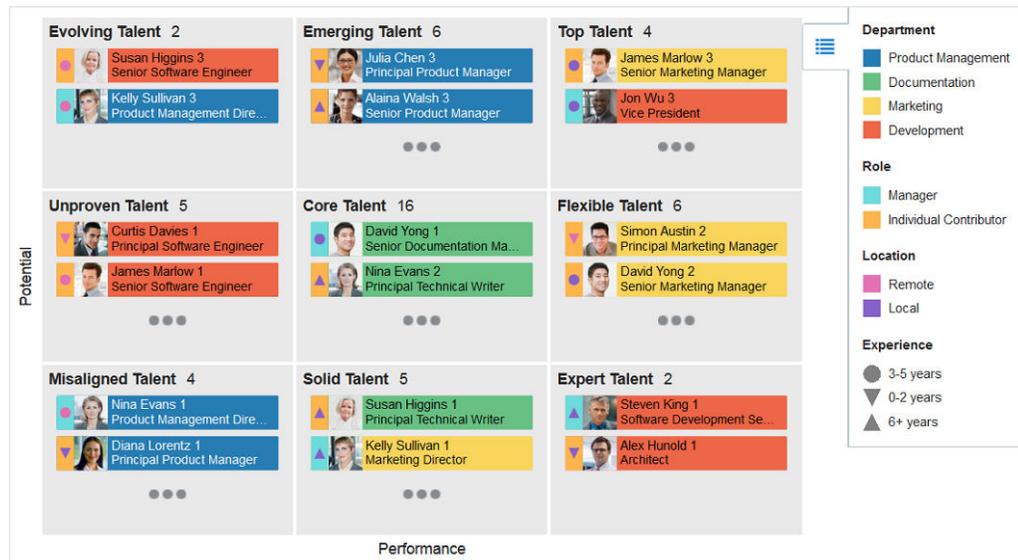
For example, as illustrated in [Figure 26-1](#) you can use the `nBox` component to compare employee potential and performance data, where the row represents employee potential and the column represents employee performance. The node that represents the employee is stamped into the appropriate cell.

Figure 26-1 NBox Component Comparing Employee Potential and Performance



NBox nodes can also be styled with colors, markers, and indicators to represent each unique value, or group, in the data set using attribute groups. Figure 26-2 shows an NBox with employee nodes styled by department, role, and experience.

Figure 26-2 NBox Nodes Styled with Attribute Groups



NBox nodes representing attribute groups can also be configured to display by size and number within each grid cell as illustrated in Figure 26-3, or across all cells are illustrated in Figure 26-4.

Figure 26-3 NBox Nodes Displayed by Size and Number Within Cells

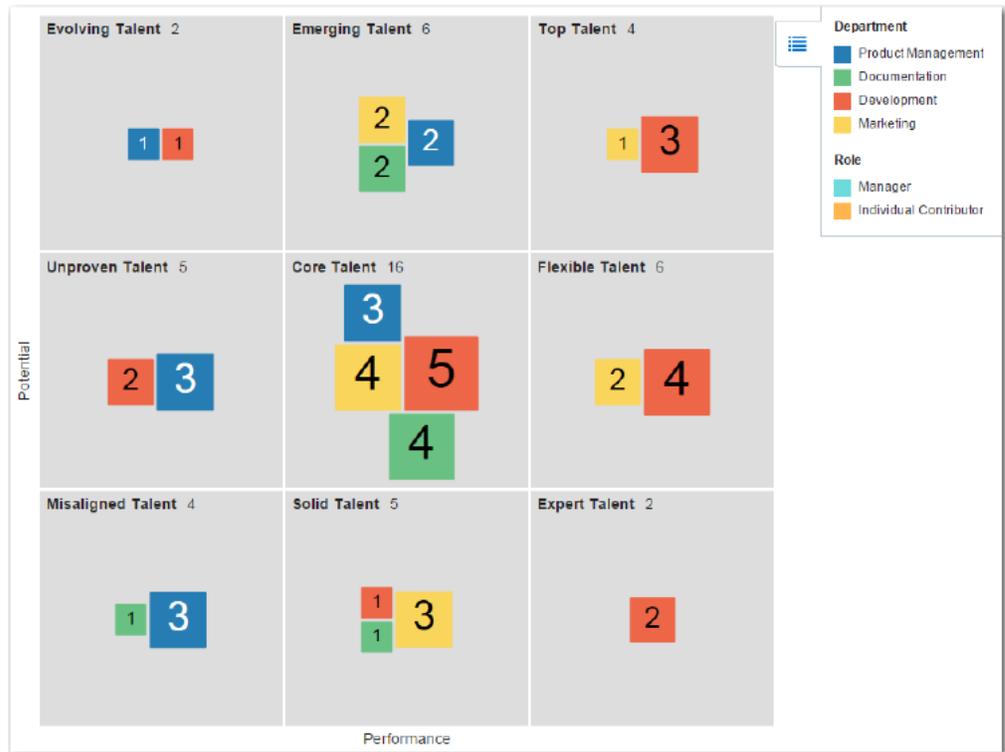
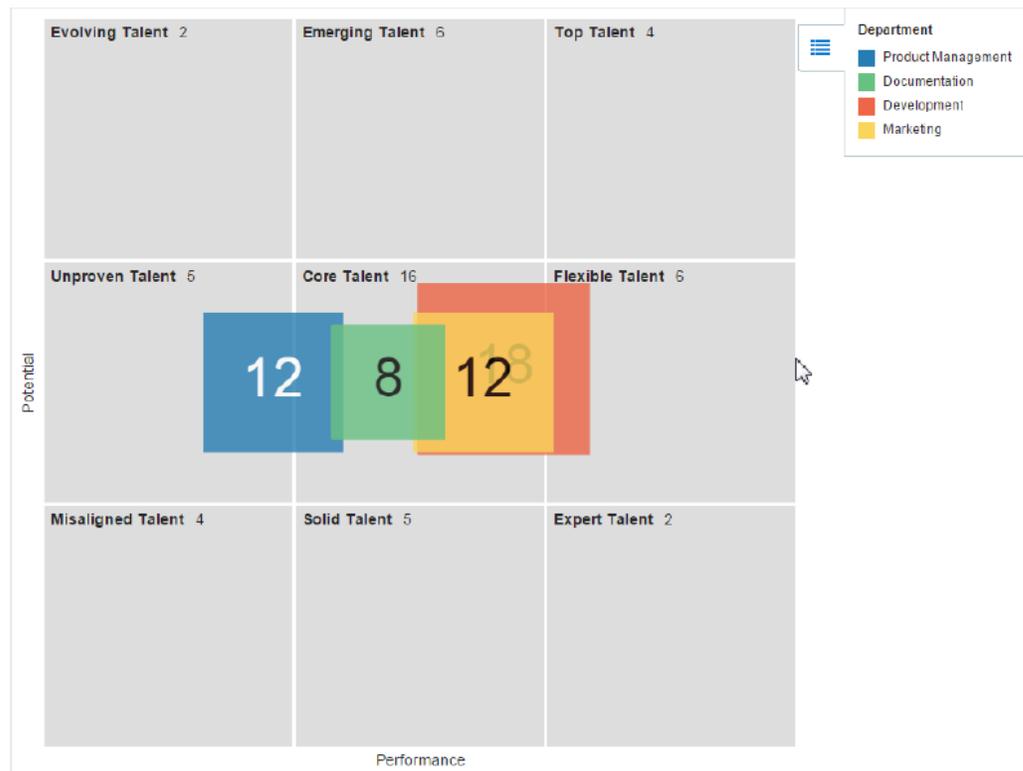


Figure 26-4 NBox Nodes Displayed by Size and Number Across Cells



End User and Presentation Features of NBoxes

The ADF Data Visualization `nBox` component provides a range of features for end users, such as selection and tooltips. They also provide a range of presentation features, such as legend display, and customizable node shapes and colors.

To use and customize NBox components, it may be helpful to understand these features and components:

- **Popup Support:** NBox components can be configured to display popup dialogs, windows, and menus that provide information or request input when the user clicks or hovers the mouse over a node.
- **Context Menus:** NBoxes support the ability to display context menus to provide additional information about the selected node.
- **Attribute Groups:** NBox nodes support the use of the `dvt:attributeGroups` tag to generate stylistic attribute values such as colors, markers, images, or indicators for each unique group in the data set.
- **Legend Support:** NBoxes display legends to provide a visual clue to the type of data controlling the image and color. If the component uses attribute groups to specify colors based on conditions such as level of performance, the legend can also display the colors used and indicate what value each color represents.
- **Node Selection Support:** NBoxes support the ability to respond to user clicks on one or more nodes to display information about the selected node(s).
- **Tooltip Support:** NBoxes support the ability to display additional information about a node when the user moves the mouse over a node.
- **Grouping:** You can configure NBoxes to group nodes together based on attribute groups and then display the group within each cell or across cells.
- **Other Node Support:** NBox components provide the ability to aggregate data if your data model includes a large number of smaller contributors in relation to the larger contributors.
- **Maximize Row or Column:** NBox rows or columns can be configured to magnify a specify row or column in the NBox, or a cell at the intersection of a specified row and column.

Additional Functionality for NBox Components

You may find it helpful to understand other ADF Faces features before you implement your NBox component. Additionally, once you have added an NBox component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that NBox components can use:

- You may want an NBox to refresh a cell to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: If enabled, users can change the way the NBox displays at runtime, and those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).

- **Accessibility:** NBox components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- **Touch devices:** When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- **Skins and styles:** You can customize the appearance of NBox components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- **Content Delivery:** You can configure your NBox to fetch data from the data source immediately upon rendering the components, or on a second request after the components have been rendered using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- **Automatic data binding:** If your application uses the Fusion technology stack, then you can create automatically bound NBoxes based on how your ADF Business Components are configured. JDeveloper provides a wizard for data binding and configuring your NBox. For more information, see the "Creating Databound NBox Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the NBox Component

To use the ADF DVT NBox component, add the NBox to a page using the Component Palette window. Then define the data for the NBox and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

NBox Data Requirements

The ADF NBox component displays data in a grid layout, with a configurable number of rows and columns used to represent two dimensions or measures of data. Nodes represent the actual data and are stamped inside the grid's cells according to where the node's value falls within the ranges or measures specified for the cells.

Data is supplied as a collections of data provided either as an implementation of the `List` interface (`java.util.ArrayList`), or a `CollectionModel` (`org.apache.myfaces.trinidad.model.CollectionModel`). The data can be of any type, typically `String`, `int`, or `long`.

The collection of NBox nodes requires an attribute that represents the unique Id for each row in the collection. The collection is mapped using a `value` attribute to stamp out each instance of the node using a component to iterate through the collection. Each time a child component is stamped out, the data for the current component is copied into a `var` property used by the data layer component in an EL Expression. Once the NBox has completed rendering, the `var` property is removed, or reverted back to its previous value. By convention, `var` is set to `node` or `link`.

The values for the `value` attribute must be stored in the node's or link's data model or in classes and managed beans if you are using UI-first development.

Configuring NBoxes

The `nBox` component has configurable attributes and child components that you can add or modify to customize the display or behavior of the NBox. The prefix `dvt:` occurs at the beginning of each NBox component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure NBox child components, attributes, and supported facets in the following areas:

- **NBox (`dvt:nBox`):** Wraps the NBox nodes. Configure the following attributes to control the NBox display:
 - `columnsTitle`: Optional label for the ordered list of columns from bottom to top along the horizontal axis of the NBox grid.
 - `rowsTitle`: Optional label for the ordered list of rows from start to end along the vertical axis of the NBox grid.
 - **Empty text (`emptyText`):** Use the `emptyText` attribute to specify the text to display if an NBox node contains no data.
 - `groupBy`: Use to group nodes together by listing the attribute group IDs in a space-separated list.
 - `groupBehavior`: Use to specify the display of a group of nodes within the cells or across the cells of the NBox.
 - **Animation:** Use a child `af:transition` tag to enable animation by setting a `triggerType` attribute to display when the NBox initially displays, and to `dataChange` when the values of the NBox nodes change.
 - **Maximize:** Use the `maximizedColumn` or `maximizedRow` attributes to magnify a specific column or row in the NBox. Specifying both attributes will magnify the cell at the intersection of the specified row and column.
 - **Other group:** Use the `otherThreshold`, and `otherColor` attributes to aggregate child data into an **Other** node.

The NBox component supports the use of these `f:facet` elements:

- **Columns:** Use to specify the number and optional label of columns for the NBox grid. An `af:group` element wraps the `dvt:nBoxColumn` elements

representing the ordered list of columns from bottom to top along the horizontal axis of the grid.

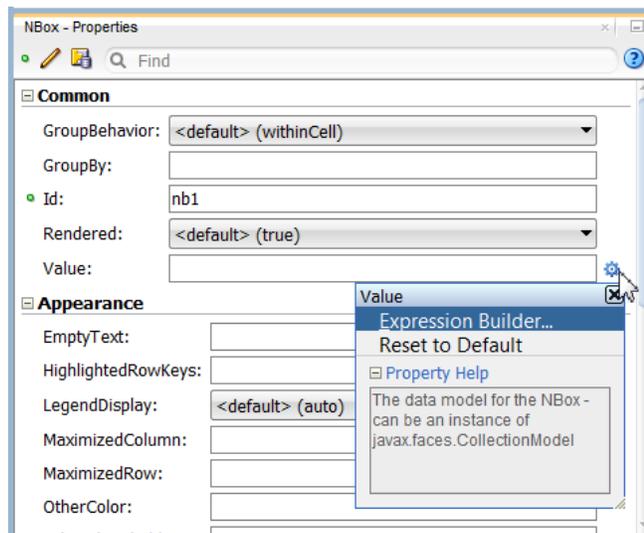
- `Rows`: Use to specify the number and optional label of rows for the NBox grid. An `af:group` element wraps the `dvt:nBoxRow` elements representing the ordered list of rows from start to end along the vertical axis of the grid.
- `cells`: Use to configure the attributes of individual cells, for example, background colors and labels. You can also use the `countLabel` and `showCount` attributes to specify a custom node count label with percentages and numbers after the cell label.
- NBox nodes (`nBoxNode`): Use to define the properties for an NBox node. The component supports the use of these `f:facet` elements:
 - `icon`: Specifies a `dvt:marker` to be used as the primary graphical element of this node, such as an employee photo or human shape.
 - `indicator`: Specifies a `dvt:marker` to be used as the secondary graphical element of this node, such as a color bar.
- Attribute groups (`attributeGroups`): Use this optional child tag of an NBox node child element, typically the `dvt:marker` component, to generate style values for each unique value, or group, in the data set.

Attribute groups are necessary to provide information for the display of the NBox and are therefore recommended.
- Legend (`legend`): Use to display multiple sections of marker and label pairs. Define the legend as a child of the NBox component.

How to Add an NBox to a Page

When you are designing your page using simple UI-first development, you use the Components window to add an NBox to a JSF page. When you drag and drop an `nBox` component onto the page, the NBox is added to your page, and you can use the Properties window to specify data values and configure additional display attributes.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. [Figure 26-5](#) shows the dropdown menu for an NBox `value` attribute.

Figure 26-5 NBox Value Attribute Dropdown Menu**Note:**

If your application uses the Fusion technology stack, then you can use data controls to create an NBox and the binding will be done for you. For more information, see the "Creating Databound NBox Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how NBox attributes and child tags can affect functionality. For more information, see [Configuring NBoxes](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for NBox Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add an NBox to a page:

1. In the ADF Data Visualization page of the Components window, from the Common panel, drag and drop an **NBox** onto the page.
2. In the Properties window, view the attributes for the NBox. Use the help button to display the complete tag documentation for the `nBox` component.
3. Expand the **Appearance** section, and enter values for the following attributes:
 - **ColumnsTitle**: Enter text to label the ordered list of columns from bottom to top along the horizontal axis of the NBox grid. Alternatively, choose **Select**

- Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
- **RowsTitle**: Enter text to label the ordered list of rows from start to end along the vertical axis of the NBox grid. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
 - **Summary**: Enter a summary of the NBox purpose and structure for accessibility support.
4. In the Structure window, right-click the **dvt:nBox** element and choose **Insert Inside NBox > Facet Columns**.
 5. In the Structure window, right-click the **f:facet - columns** element and choose **Insert inside Facet columns > Group**.
 6. In the Structure window, right-click the **Group** element and for each column you wish to add to the NBox, choose **Insert inside Group > ADF Data Visualizations > NBox Column**.
 7. In the Insert NBox Column dialog enter the value representing a defined range of data within the column dimension of the NBox. For example, for an NBox displaying a range of US population size across 3 columns, you might enter `low`, `medium`, and `high`.
 8. In the Structure window, right-click the **dvt:nBox** element and choose **Insert Inside NBox > Facet Rows**.
 9. In the Structure window, right-click the **f:facet - row** element and choose **Insert inside Facet Rows > Group**.
 10. In the Structure window, right-click the **Group** elements and for each row you wish to add to the NBox, choose **Insert inside Group > ADF Data Visualizations > NBox Row**.
 11. In the Insert NBox Row dialog enter the value representing a defined range of data within the row dimension of the NBox. For example, for an NBox displaying a range of income levels across 3 rows, you might enter `lower`, `medium`, and `upper`.

What Happens When You Add an NBox to a Page

Developer generates only a minimal set of tags when you drag and drop an NBox from the Components window onto a JSF page and choose not to bind the data during creation.

The generated code is:

```
<dvt:nBox id="nb1" />
```

If you choose to bind the data to a data control when creating the NBox, JDeveloper generates code based on the data model. For more information, see the "Creating Databound NBox Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Using Pivot Table Components

This chapter describes how to use the ADF Data Visualization `pivotTable` and `pivotFilterBar` components to display data in pivot tables using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the Fusion technology stack, then you can also use data controls to create pivot tables. JDeveloper provides a wizard for data binding and configuring your pivot table. For more information, see the "Creating Databound Pivot Table and Pivot Filter Bar Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Pivot Table Component](#)
- [Using the Pivot Table Component](#)
- [Configuring Header and Data Cell Stamps](#)
- [Using Pivot Filter Bars](#)
- [Adding Interactivity to Pivot Tables](#)
- [Formatting Pivot Table Cell Content With CellFormat](#)

About the Pivot Table Component

ADF DVT Pivot tables display data in a grid layout with unlimited layers of hierarchically nested row header cells and column header cells. Similar to spreadsheets, pivot tables provide the option of automatically generating subtotals and totals for grid data.

A pivot table allows users to pivot or reposition row or column header data labels and the associated data layer from one location on the row or column edge to another to obtain different views of data, supporting interactive analysis.

A pivot filter bar is a component that can be added to a pivot table to provide the user with a way to filter pivot table data in layers not displayed in one of the row or column edges of the pivot table. Users can also drag and drop these layers between the pivot filter bar and the associated pivot table to change the view of the data. A pivot filter bar can also be used to change the graphical display of data in a graph.

Pivot Table and Pivot Filter Bar Component Use Cases and Examples

A pivot table displays a grid of data with rows and columns. [Figure 27-1](#) shows a pivot table with multiple attributes nested on its rows and columns.

Figure 27-1 Sales Pivot Table with Multiple Rows and Columns

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20,000.000	500.000	200.000	50.000
	Canoes	15,000.000	1,500.000	75.000	8.000
2006	Tents	10,000.000	250.000	100.000	25.000
	Canoes	7,500.000	750.000	40.000	4.000
2005	Tents	5,000.000	125.000	50.000	15.000
	Canoes	3,750.000	375.000	20.000	2.000

Pivot table data cells support other data display components such as sparkcharts, gauges, and graphs. [Figure 27-2](#) shows a pivot table with sparkcharts illustrating data trends over time in a data cell.

Figure 27-2 Pivot Table with Sparkcharts Stamped in Data Cells

		Sales						Units							
		All Channels						All Channels							
		2005	2006	2007	2008	2009	2010	Trend	2005	2006	2007	2008	2009	2010	Trend
Tents	World	5000.0	10000.0	20000.0	8000.0	18000.0	30000.0		50.0	100.0	200.0	70.0	1900.0	300.0	
	Boston	125.0	250.0	500.0	200.0	400.0	900.0		15.0	25.0	50.0	22.0	40.0	110.0	
Canoes	World	3750.0	7500.0	15000.0	5000.0	12000.0	29000.0		20.0	40.0	75.0	35.0	70.0	160.0	
	Boston	375.0	750.0	1500.0	450.0	1100.0	3100.0		2.0	4.0	8.0	3.0	7.0	16.0	

[Figure 27-3](#) shows a pivot table with graphs stamped in data cells.

Figure 27-3 Pivot Table with Graphs Stamped in Data Cells

		Sales				Units			
		All Channels		Channel Total		All Channels		Channel Total	
		World	Boston	Geography Total	Channel Total	World	Boston	Geography Total	Channel Total
2007	Tents								
	Canoes								
Product Total									

Header and data cells in pivot tables can be customized to display image, icons or links, and to display stoplight and conditional formatting. [Figure 27-4](#) shows a pivot table with conditional formatting to display levels of sales performance.

Figure 27-4 Conditional Data Cell Formatting

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20000.0	500.0	200.0	50.0
	Canoes	15000.0	1500.0	75.0	8.0
2006	Tents	10000.0	250.0	100.0	25.0
	Canoes	7500.0	750.0	40.0	4.0
2005	Tents	5000.0	125.0	50.0	15.0
	Canoes	3750.0	375.0	20.0	2.0

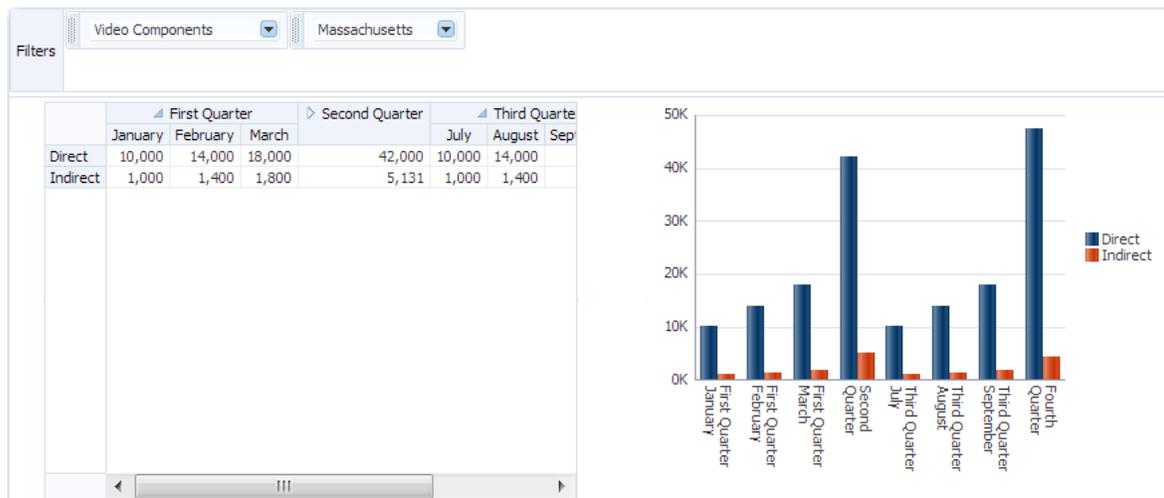
A pivot filter bar is a component that can be associated with a pivot table to provide the user with a way to filter pivot table data in layers not displayed in the row or column edges of the pivot table. Users can also drag and drop these layers between the pivot filter bar and the associated pivot table to change the view of the data. [Figure 27-5](#) shows a pivot filter bar associated with a pivot table.

Figure 27-5 Pivot Filter Bar Component Associated with Pivot Table

Filters		Month	Quarter
		January	First Quarter
		Massachusetts	Rhode Island
Direct	Audio Components	5,000	2,000
	Video Components	10,000	500
	Gaming Systems	10,000	700
	Photography Equipment	7,000	3,050
	Phones	10,500	1,050
	Miscellaneous	10,500	1,050
Indirect	Audio Components	500	200
	Video Components	1,000	50
	Gaming Systems	1,000	70
	Photography Equipment	700	305
	Phones	1,050	105
	Miscellaneous	1,000	105

A pivot filter bar can also be used to change the display of data in a graph associated with the pivot table. [Figure 27-6](#) shows a filtered view of quarterly sales data displayed simultaneously in a pivot table and on a bar graph.

Figure 27-6 Pivot Filter Bar Associated with Pivot Table and Graph



End User and Presentation Features of Pivot Table Components

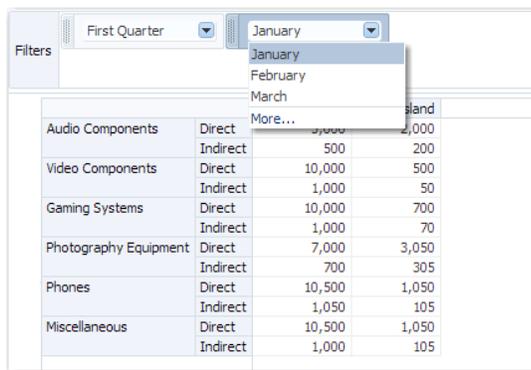
The ADF Data Visualization pivot table component provides a range of features for end users, such as pivoting, sorting columns, and selection of one or more rows, columns, or cells, and then executing an application defined action on the selection. It also provides a range of presentation features, such as unlimited layers of hierarchically nested row header and column header cells.

Pivot Filter Bar

The data filtering capacity in a pivot table can be enhanced with an optional pivot filter bar. Zero or more layers of data not already displayed in the pivot table row edge or column edge are displayed in the page edge.

Figure 27-7 shows a pivot filter bar with Quarter and Month layers that can be used to filter the data displayed in the pivot table.

Figure 27-7 Pivot Filter Bar with Data Layer Filters



		January	February	March	April
Audio Components	Direct	5,000	2,000		
	Indirect	500	200		
Video Components	Direct	10,000	500		
	Indirect	1,000	50		
Gaming Systems	Direct	10,000	700		
	Indirect	1,000	70		
Photography Equipment	Direct	7,000	3,050		
	Indirect	700	305		
Phones	Direct	10,500	1,050		
	Indirect	1,050	105		
Miscellaneous	Direct	10,500	1,050		
	Indirect	1,000	105		

Pivoting

You can drag any layer in a pivot table to a different location on the same edge, to the opposite edge, or to the associated pivot filter bar (if present), to change the view of the data in the pivot table. Any layer in a pivot filter bar can be dragged to a different location within the pivot filter bar, or to the row or column edge of the pivot table. This operation is called pivoting and is enabled by default.

When you move the mouse over a layer, the layer's pivot handle and an optional pivot label are displayed. If you move the mouse over the pivot handle, the cursor changes to a four-point arrow drag cursor. You can then use the handle to drag the layer to the new location. If you move the mouse over a layer on the row edge, the pivot handle appears above the layer, as shown in Figure 27-8.

Figure 27-8 Display of Pivot Handle on the Row Edge

		Sales		Units		
		All Channels		All Channels		
		World	Boston	World	Boston	
Time						
2007	Tents	20,000	500	200	50	
	Canoes	15,000	1,500	75	8	
2006	Tents	10,000	250	100	25	
	Canoes	7,500	750	40	4	
2005	Tents	5,000	125	50	15	
	Canoes	3,750	375	20	2	

If you move the cursor over a layer in the column edge, the pivot handle appears to the left of the layer, as shown in [Figure 27-9](#).

Figure 27-9 Display of Pivot Handle on the Column Edge

		Sales		Units		
		All Channels		All Channels		
		World	Boston	World	Boston	
Channel						
2007	Tents	20,000	500	200	50	
	Canoes	15,000	1,500	75	8	
2006	Tents	10,000	250	100	25	
	Canoes	7,500	750	40	4	
2005	Tents	5,000	125	50	15	
	Canoes	3,750	375	20	2	

If, in [Figure 27-8](#), you drag the pivot handle of the Time (Year) layer from the row edge to the column edge between the Measure (Sales) layer and the Channel layer, the pivot table will change shape as shown in [Figure 27-10](#).

Figure 27-10 Sales Pivot Table After Pivot of Year

	Sales						Units					
	2007		2006		2005		2007		2006		2005	
	All Channels											
	World	Boston										
Tents	20,000	500	10,000	250	5,000	125	200	50	100	25	50	15
Canoes	15,000	1,500	7,500	750	3,750	375	75	8	40	4	20	2

You can customize pivoting to disable pivot labels and pivoting. If both are disabled, the pivot handle does not display when mousing over the layer.

Editing Data Cells

Pivot tables can contain both read-only and editable data cells. Editable cells are those containing an input component, for example, `af:inputText` or `af:comboBox`.

When a pivot table containing editable cells is initially displayed, the first data cell is selected and the pivot table is open for editing. Users can initiate editing anywhere in the pivot table by clicking in a cell to edit or overwrite the cell value. Clicking in editable cells enables the user to identify a specific location within the cell, and then navigate within that cell using the arrow keys. Any edit performed on an editable cell can be reverted by pressing Esc.



Note:

Pressing Esc will not revert a value selected in a list of values component, for example `af:inputComboboxListOfValues`, since the value is submitted immediately when the dropdown is closed.

Figure 27-11 shows a pivot table data cell open for direct editing.

Figure 27-11 Data Cell Open for Direct Editing

				Sales	Units	Available	Price	Color	Weight	Link	Size	Supply Date
2007	Tents	All Channels	World	20000.0	200.0					main-link	L	1/1/2000
			Boston	500.0	50.0						main-link	S
	Jacket	All Channels	World	40000.0	4000.0	Item Available	11.0	red	44.0	Main-link	L	1/1/2000
			Boston	700.0	70.0	Item Available	13.0	coffee	42.0	Main-link	S	4/4/2004
2008	Tents	All Channels	World	30000.0	300.0	Item Available	36.0	red	33.0	Main-link	L	1/1/2000
			Boston	3000.0	30.0	Item Available	69.0	coffee	66.0	Main-link	S	4/4/2004
	Jacket	All Channels	World	10000.0	1000.0	Item Available	12.0	red	44.0	Sub-link	M	3/3/2003
			Boston	600.0	60.0	Item Available	15.0	coffee	42.0	Main-link	S	4/4/2004

Data cells selected for dropdown list editing are displayed as shown in Figure 27-12.

Figure 27-12 Data Cell Open for Dropdown List Editing

				Sales	Units	Available	Price	Color	Weight	Link	Size	Supply Date
2007	Tents	All Channels	World	20000.0	200.0	Item Available	33.0	red	33.0	Main-link	L	1/1/2000
			Boston	500.0	50.0	Item Available	66.0	coffee	66.0	Main-link	S	4/4/2004
	Jacket	All Channels	World	40000.0	4000.0	Item Available	11.0	red	44.0	Main-link	L	1/1/2000
			Boston	700.0	70.0	Item Available	13.0	red	42.0	Main-link	S	4/4/2004
2008	Tents	All Channels	World	30000.0	300.0	Item Available	36.0	coffee	33.0	Main-link	L	1/1/2000
			Boston	3000.0	30.0	Item Available	69.0	coffee	66.0	Main-link	S	4/4/2004
	Jacket	All Channels	World	10000.0	1000.0	Item Available	12.0	red	44.0	Sub-link	M	3/3/2003
			Boston	600.0	60.0	Item Available	15.0	coffee	42.0	Main-link	S	4/4/2004

While in editing mode, you can navigate through pivot table data cells using Tab or Enter. To quickly navigate to the cell below or above the currently selected cell, use Ctrl+arrow keys. When using the Enter key to navigate, an active link will automatically be launched for a cell containing an active link. When using Tab or Shift+Tab to navigate, data cells containing multiple editable components, as in the case of both an `af:inputDate` and date picker in the same cell, the Tab highlights each editable component in turn. When tabbing through the last column of the pivot table, the first column of the next row is highlighted, and when Shift-Tabbing through the first column in the pivot table, the last column of the previous row is highlighted.

Once editing mode is initiated, users can navigate through read-only data cells to editable data cells, maintaining the editing mode. While an editable cell is selected, you can select other cells using Ctrl or Shift+click without enabling editing in the new cells and maintaining editing in the original cell.

 **Note:**

- You can customize the submission of the modified data cell values. The `ValueSubmitPolicy` attribute has two possible values, `Immediate` and `Lazy`. With `Immediate`, the value is instantly submitted when the user navigates to another cell. With `Lazy`, the value is not submitted until the user navigates to another cell that requires a fetch, or the user performs an action that sends a request to the server to submit the updated values.
- In order to temporarily or permanently write values back to a set of cells within a cube, called a writeback, the pivot table must be bound to a data control or data model that supports writeback operations. A pivot table row set based data control is transformed into a cube that supports writeback operations.

Data and Header Sorting

Pivot tables support sorting of data within the pivot table. When sorting is enabled, ascending and descending sort icons are displayed as the user hovers the mouse over the innermost layer of the column header.

By default, the `sortMode` attribute of the `pivotTable` component is set to `grouped`, effectively sorting the data grouped by the second-to-innermost layer of the row edge. [Figure 27-13](#) shows the data in the World Sales column sorted descending, where the products within each year are grouped and thereby also sorted descending.

Figure 27-13 Ascending and Descending Sorting Icons in a Pivot Table

		Sales		Units	
		All Channels		All Channels	
		World 	Boston	World	Boston
2007	Tents	20,000	500	200	50
	Canoes	15,000	1,500	75	8
2006	Tents	10,000	250	100	25
	Canoes	7,500	750	40	4
2005	Tents	5,000	125	50	15
	Canoes	3,750	375	20	2

You can also sort data display by column and row headers using context menu options. Setting the sort order on the column or row headers configures all the columns and rows in that layer to be similarly sorted. [Figure 27-14](#) shows a pivot table with the US City column headers sorted Left to Right with a context menu option to change the sort order to Right to Left.

Figure 27-14 Pivot Table Column Header Sorting

Car	Salesperson	Measure US City	Acres						Av	
			Atlanta		Boston		Atlanta			
	Product	Canoes	Bobt							
Opel	Ezra	2,402	2,390	2,357	3,156	3,144	3,111	2,247	2,235	
	Daphne	2,802	2,756	2,723	2,757	3,556	3,544	3,511	2,647	2,635
	Chet	2,138	2,154	2,121	2,093	2,892	2,880	2,847	1,983	1,971
	Betty	2,420	2,420	2,420	2,375	3,174	3,162	3,129	2,265	2,253
	Archie	2,768	2,756	2,723	3,522	3,510	3,477	2,613	2,601	
Nova	Ezra	2,166	2,154	2,121	2,920	2,908	2,875	2,011	1,999	
	Daphne	2,566	2,554	2,521	3,320	3,308	3,275	2,411	2,399	
	Chet	1,902	1,890	1,857	2,656	2,644	2,611	1,747	1,735	
	Betty	2,184	2,172	2,139	2,938	2,926	2,893	2,029	2,017	
	Archie	2,532	2,520	2,487	3,286	3,274	3,241	2,377	2,365	

Figure 27-15 shows a pivot table with the Car row sorted with headers from Bottom to Top with a context menu option or change the sort to Top to Bottom.

Figure 27-15 Pivot Table Row Header Sorting

Car	Salesperson	Measure US City	Acres						Av	
			Atlanta		Boston		Atlanta			
	Product	Canoes	Bobbins	Avocados	Canoes	Bobbins	Avocados	Canoes	Bobbins	
Opel	Ezra	2,402	2,390	2,357	3,156	3,144	3,111	2,247	2,235	
	Daphne	2,802	2,756	2,723	2,757	3,556	3,544	3,511	2,647	2,635
	Chet	2,138	2,154	2,121	2,093	2,892	2,880	2,847	1,983	1,971
	Betty	2,420	2,420	2,420	2,375	3,174	3,162	3,129	2,265	2,253
	Archie	2,768	2,756	2,723	3,522	3,510	3,477	2,613	2,601	
Nova	Ezra	2,166	2,154	2,121	2,920	2,908	2,875	2,011	1,999	
	Daphne	2,566	2,554	2,521	3,320	3,308	3,275	2,411	2,399	
	Chet	1,902	1,890	1,857	2,656	2,644	2,611	1,747	1,735	
	Betty	2,184	2,172	2,139	2,938	2,926	2,893	2,029	2,017	
	Archie	2,532	2,520	2,487	3,286	3,274	3,241	2,377	2,365	
McLaren	Ezra	2,570	2,558	2,525	3,324	3,312	3,279	2,415	2,403	
	Daphne	2,970	2,958	2,925	3,724	3,712	3,679	2,815	2,803	
	Chet	2,306	2,294	2,261	3,060	3,048	3,015	2,151	2,139	
	Betty	2,588	2,576	2,543	3,342	3,330	3,297	2,433	2,421	
	Archie	2,936	2,924	2,891	3,690	3,678	3,645	2,781	2,769	

Drilling

Pivot tables support two types of drilling including insert drilling, and filter drilling. With insert drilling, the expand operation reveals the detail data while preserving the sibling and aggregate data. With filter drilling, the expand operation displays the detail data only, filtering out sibling and aggregate data.

For example, Figure 27-16 and Figure 27-17 illustrate how drilling is used to display product data within each year; revealing that the 2007 total sales number of 52,500 is composed of 25,500 for tents and 27,000 for canoes. This total contributes to the aggregated total of all sales for all years of 128,172. Figure 27-16 shows a pivot table using insert drilling with the total number of 52,500 displayed alongside the detail numbers. The data for other years and the aggregated total for all years is also available.

Figure 27-16 Pivot Table with Insert Drilling Enabled

	Sales	Units		Sales	Units
TOTAL	128,172	1,135		128,172	1,135
▶ 2007	52,500	410		52,500	410
▶ 2006	54,150				
▶ 2005	21,522				
			TOTAL	128,172	1,135
			▼ 2007		
			Tents	25,500	275
			Canoes	27,000	135
			▼ 2006	54,150	507
			Tents	30,750	375
			Canoes	23,400	132
			▶ 2005	21,522	218

Figure 27-17 shows a pivot table using filter drilling with only the detail numbers are displayed. The numbers for other years, and the aggregated total for all years is filtered out.

Figure 27-17 Pivot Table with Filter Drilling Enabled

	Sales	Units		Sales	Units
TOTAL	128,172	1,135			
▶ 2007	52,500	410			
▶ 2006	54,150				
▶ 2005	21,522				
			TOTAL	52,500	410
			▼ 2007		
			Tents	25,500	275
			Canoes	27,000	135

At runtime, a drill icon is enabled in the parent attribute display label for both types of drilling.

If you do not perform a pivot operation, then the drill operation will remain for the life of the session. However, in the case of pivoting a drilled child attribute away from a parent attribute, you can configure the desired behavior using Oracle MDS (Metadata Services) customization. For information about creating customizable applications using MDS, see the "Customizing Applications with MDS" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Scrolling and Page Controls

Pivot tables support both on-demand data scrolling and fixed page controls in order to support large data sets while maintaining performance.

While scrolling, only the data that is scrolled into view in the pivot table is loaded. As the user scrolls vertically or horizontally, data is fetched for the portion of the pivot table that has scrolled into view, and data that is no longer needed is discarded.

Figure 27-18 shows a pivot table with a large data set using on-demand data scrolling.

Figure 27-18 On-Demand Data Scrolling in a Pivot Table

		Acres						Bytes			
		Atlanta			Boston			Atlanta			
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados
Harley-Davidson	Daphne	2,870	2,903	2,915	3,624	3,657	3,669	2,715	2,748	2,760	
	Ezra	2,470	2,503	2,515	3,224	3,257	3,269	2,315	2,348	2,360	
	Archie	2,831	2,864	2,876	3,585	3,618	3,630	2,676	2,709	2,721	
	Betty	2,483	2,516	2,528	3,237	3,270	3,282	2,328	2,361	2,373	
	Chet	2,201	2,234	2,246	2,955	2,988	3,000	2,046	2,079	2,091	
Isdera	Daphne	2,865	2,898	2,910	3,619	3,652	3,664	2,710	2,743	2,755	
	Ezra	2,465	2,498	2,510	3,219	3,252	3,264	2,310	2,343	2,355	
	Archie	2,929	2,962	2,974	3,683	3,716	3,728	2,774	2,807	2,819	
	Betty	2,581	2,614	2,626	3,335	3,368	3,380	2,426	2,459	2,471	
	Chet	2,299	2,332	2,344	3,053	3,086	3,098	2,144	2,177	2,189	
Jaguar	Daphne	2,963	2,996	3,008	3,717	3,750	3,762	2,808	2,841	2,853	
	Ezra	2,563	2,596	2,608	3,317	3,350	3,362	2,408	2,441	2,453	
	Archie	2,917	2,950	2,962	3,671	3,704	3,716	2,762	2,795	2,807	
	Betty	2,569	2,602	2,614	3,323	3,356	3,368	2,414	2,447	2,459	
	Chet	2,287	2,320	2,332	3,041	3,074	3,086	2,132	2,165	2,177	
Kia	Daphne	2,951	2,984	2,996	3,705	3,738	3,750	2,796	2,829	2,841	
	Ezra	2,551	2,584	2,596	3,305	3,338	3,350	2,396	2,429	2,441	
	Archie	2,790	2,823	2,835	3,544	3,577	3,589	2,635	2,668	2,680	
	Betty	2,442	2,475	2,487	3,196	3,229	3,241	2,287	2,320	2,332	
	Chet	2,160	2,193	2,205	2,914	2,947	2,959	2,005	2,038	2,050	
	Daphne	2,824	2,857	2,869	3,578	3,611	3,623	2,669	2,702	2,714	

Instead of scrollbars, you can configure a page control to navigate large data sets in pivot tables for desktop applications and for mobile browsers on touch devices. For example, the page control for columns display at the top of the pivot table and the page control for rows displays at the foot of the pivot table as shown in Figure 27-19.

Figure 27-19 Pivot Table Column and Row Page Controls

		Columns 1 - 9 of 54									
		Acres						Bytes			
		Atlanta			Boston			Atlanta			
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados
Aston Martin	Archie	2,449	2,482	2,494	3,203	3,236	3,248	2,294	2,327	2,339	3,041
	Betty	2,101	2,134	2,146	2,855	2,888	2,900	1,946	1,979	1,991	2,701
	Chet	1,819	1,852	1,864	2,573	2,606	2,618	1,664	1,697	1,709	2,411
	Daphne	2,483	2,516	2,528	3,237	3,270	3,282	2,328	2,361	2,373	3,083
	Ezra	2,083	2,116	2,128	2,837	2,870	2,882	1,928	1,961	1,973	2,685
Bentley	Archie	2,522	2,555	2,567	3,276	3,309	3,321	2,367	2,400	2,412	3,113
	Betty	2,174	2,207	2,219	2,928	2,961	2,973	2,019	2,052	2,064	2,765
	Chet	1,892	1,925	1,937	2,646	2,679	2,691	1,737	1,770	1,782	2,483
	Daphne	2,556	2,589	2,601	3,310	3,343	3,355	2,401	2,434	2,446	3,113
	Ezra	2,156	2,189	2,201	2,910	2,943	2,955	2,001	2,034	2,046	2,747
Corvette	Archie	2,667	2,700	2,712	3,421	3,454	3,466	2,512	2,545	2,557	3,267
	Betty	2,319	2,352	2,364	3,073	3,106	3,118	2,164	2,197	2,209	2,919
	Chet	2,037	2,070	2,082	2,791	2,824	2,836	1,882	1,915	1,927	2,637
	Daphne	2,701	2,734	2,746	3,455	3,488	3,500	2,546	2,579	2,591	3,301
	Ezra	2,301	2,334	2,346	3,055	3,088	3,100	2,146	2,179	2,191	2,901
DeLorean	Archie	2,647	2,680	2,692	3,401	3,434	3,446	2,492	2,525	2,537	3,247
	Betty	2,299	2,332	2,344	3,053	3,086	3,098	2,144	2,177	2,189	2,899
	Chet	2,017	2,050	2,062	2,771	2,804	2,816	1,862	1,895	1,907	2,617
	Daphne	2,681	2,714	2,726	3,435	3,468	3,480	2,526	2,559	2,571	3,281

By default, on desktop devices, tables render a scroll bar that allows the user to scroll through all rows. On tablet devices, instead of a scroll bar, the table is paginated and displays a footer that allows the user to jump to specific pages of rows. You can change the default (auto) value by setting the ScrollPolicy attribute to either scroll or page based on the type of device.

You can customize the initial display of the pivot table by specifying the starting visible row or column data cell or header layer. Use `startRow` and `startColumn` attributes to specify the first visible row and column of data. Use `rowHeaderStartLayer` and `columnHeaderStartLayer` attributes to specify the first visible row or header layer. Upon initial display of the pivot table, the scrollbar or page control will automatically be positioned for these attribute settings.

Persistent Header Layers

You can configure pivot tables to always display the labels that appear above each row header layer and beside each column header layer.

This is useful when displaying large data sets to keep the column and row header labels in view with the data. To configure persistent display of the row and column header labels for the `pivotTable` component, set the `layerLabelMode` attribute to `rendered`.

Split View of Large Data Sets

Pivot tables displaying large data sets can be configured to support a user defined split view of the data. In a split view the pivot table is split into multiple panes vertically and/or horizontally, facilitating a side-by-side viewing of rows or columns not located next to each in the table.

When enabled, a listener is notified after a split is successfully added or removed from the pivot table. For example, you might want to keep the aggregate level year information viewable while scrolling through the weeks at the end of the year at the same time.

By default, the option to split or unsplit a view of the data is available from any pivot table header or data cell context menu. Users can split columns, rows, or rows and columns to define the viewable panes of the pivot table. The portion of the available space allocated to each pane is determined by the scroll position of the cell on which the **Split View** command is invoked.

To split only columns, select the column header cell for the column that should be the first column of the second pane, and in the context menu select **Split View**. [Figure 27-20](#) shows a columns only split view pivot table data.

Figure 27-20 Column Only Split View of Pivot Table Data

		Acres						Byte			
		Atlanta			Boston			Atlanta			
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	
Ferrari	Archie	3,268	3,301	3,313	4,022	4,055	4,067	3,113	3,146	3,158	
	Betty	2,920	2,953	2,965	3,674	3,707	3,719	2,765	2,798	2,810	
	Chet	2,638	2,671	2,683	3,392	3,425	3,437	2,483	2,516	2,528	
	Daphne	3,302	3,335	3,347	4,056	4,089	4,101	3,147	3,180	3,192	
	Ezra	2,902	2,935	2,947	3,656	3,689	3,701	2,747	2,780	2,792	
Geo	Archie	2,836	2,869	2,881	3,590	3,623	3,635	2,681	2,714	2,726	
	Betty	2,488	2,521	2,533	3,242	3,275	3,287	2,333	2,366	2,378	
	Chet	2,206	2,239	2,251	2,960	2,993	3,005	2,051	2,084	2,096	
	Daphne	2,870	2,903	2,915	3,624	3,657	3,669	2,715	2,748	2,760	
	Ezra	2,470	2,503	2,515	3,224	3,257	3,269	2,315	2,348	2,360	
Harley-Davidson	Archie	2,831	2,864	2,876	3,585	3,618	3,630	2,676	2,709	2,721	
	Betty	2,483	2,516	2,528	3,237	3,270	3,282	2,328	2,361	2,373	
	Chet	2,201	2,234	2,246	2,955	2,988	3,000	2,046	2,079	2,091	
	Daphne	2,865	2,898	2,910	3,619	3,652	3,664	2,710	2,743	2,755	
	Ezra	2,465	2,498	2,510	3,219	3,252	3,264	2,310	2,343	2,355	
Isdera	Archie	2,929	2,962	2,974	3,683	3,716	3,728	2,774	2,807	2,819	
	Betty	2,581	2,614	2,626	3,335	3,368	3,380	2,426	2,459	2,471	
	Chet	2,299	2,332	2,344	3,053	3,086	3,098	2,144	2,177	2,189	
	Daphne	2,963	2,996	3,008	3,717	3,750	3,762	2,808	2,841	2,853	
	Ezra	2,563	2,596	2,608	3,317	3,350	3,362	2,408	2,441	2,453	
Jaguar	Archie	2,917	2,950	2,962	3,671	3,704	3,716	2,762	2,795	2,807	

To split only rows, select the row header cell for the row that should be the first row of the second pane, and in the context menu select **Split View**. Figure 27-21 shows a row only split view of pivot table data.

Figure 27-21 Row Only Split View of Pivot Table Data

		Acres						Bytes			
		Atlanta			Boston			Atlanta			
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	
Jaguar	Archie	2,917	2,950	2,962	3,671	3,704	3,716	2,762	2,795	2,807	
	Betty	2,569	2,602	2,614	3,323	3,356	3,368	2,414	2,447	2,459	
	Chet	2,287	2,320	2,332	3,041	3,074	3,086	2,132	2,165	2,177	
	Daphne	2,951	2,984	2,996	3,705	3,738	3,750	2,796	2,829	2,841	
	Ezra	2,551	2,584	2,596	3,305	3,338	3,350	2,396	2,429	2,441	
Kia	Archie	2,790	2,823	2,835	3,544	3,577	3,589	2,635	2,668	2,680	
	Betty	2,442	2,475	2,487	3,196	3,229	3,241	2,287	2,320	2,332	
	Chet	2,160	2,193	2,205	2,914	2,947	2,959	2,005	2,038	2,050	
	Daphne	2,824	2,857	2,869	3,578	3,611	3,623	2,669	2,702	2,714	
	Ezra	2,424	2,457	2,469	3,178	3,211	3,223	2,269	2,302	2,314	
Lamborghini	Archie	3,233	3,266	3,278	3,987	4,020	4,032	3,078	3,111	3,123	
	Betty	2,885	2,918	2,930	3,639	3,672	3,684	2,730	2,763	2,775	
	Chet	2,603	2,636	2,648	3,357	3,390	3,402	2,448	2,481	2,493	
	Daphne	3,267	3,300	3,312	4,021	4,054	4,066	3,112	3,145	3,157	
	Ezra	2,867	2,900	2,912	3,621	3,654	3,666	2,712	2,745	2,757	
Opel	Archie	2,801	2,834	2,846	3,545	3,578	3,590	2,736	2,769	2,781	
	Betty	2,375	2,408	2,420	3,129	3,162	3,174	2,220	2,253	2,265	
	Chet	2,093	2,126	2,138	2,847	2,880	2,892	1,938	1,971	1,983	
	Daphne	2,757	2,790	2,802	3,511	3,544	3,556	2,602	2,635	2,647	
	Ezra	2,357	2,390	2,402	3,111	3,144	3,156	2,202	2,235	2,247	

To split both rows and columns, select the data cell that should be the first cell of the last pane, and in the context menu select **Split View**. Figure 27-22 shows a row and column split view of pivot table data.

Figure 27-22 Row and Column Split View of Pivot Table Data

		Decibels						Inches		
		Atlanta			Atlanta			Boston		
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes
Aston Martin	Archie	2,228	2,261	2,273	1,939	1,972	1,984	2,693	2,726	2,738
	Betty	1,880	1,913	1,925	1,591	1,624	1,636	2,345	2,378	2,390
	Chet	1,598	1,631	1,643	1,309	1,342	1,354	2,063	2,096	2,108
	Daphne	2,262	2,295	2,307	1,973	2,006	2,018	2,727	2,760	2,772
	Ezra	1,862	1,895	1,907	1,573	1,606	1,618	2,327	2,360	2,372
Bentley	Archie	2,301	2,334	2,346	2,012	2,045	2,057	2,766	2,799	2,811
	Betty	1,953	1,986	1,998	1,664	1,697	1,709	2,418	2,451	2,463
	Chet	1,671	1,704	1,716	1,382	1,415	1,427	2,136	2,169	2,181
	Daphne	2,335	2,368	2,380	2,046	2,079	2,091	2,800	2,833	2,845
Corvette	Ezra	1,935	1,968	1,980	1,646	1,679	1,691	2,400	2,433	2,445
	Archie	2,446	2,479	2,491	2,157	2,190	2,202	2,911	2,944	2,956
	Betty	2,098	2,131	2,143	1,809	1,842	1,854	2,563	2,596	2,608
	Chet	1,816	1,849	1,861	1,527	1,560	1,572	2,281	2,314	2,326
DeLorean	Daphne	2,480	2,513	2,525	2,191	2,224	2,236	2,945	2,978	2,990
	Ezra	2,080	2,113	2,125	1,791	1,824	1,836	2,545	2,578	2,590
	Archie	2,426	2,459	2,471	2,137	2,170	2,182	2,891	2,924	2,936
	Betty	2,078	2,111	2,123	1,789	1,822	1,834	2,543	2,576	2,588
Ezra	Chet	1,796	1,829	1,841	1,507	1,540	1,552	2,261	2,294	2,306
	Daphne	2,460	2,493	2,505	2,171	2,204	2,216	2,925	2,958	2,970
	Ezra	2,060	2,093	2,105	1,771	1,804	1,816	2,525	2,558	2,570
	Archie	2,045	2,078	2,090	1,755	1,788	1,800	2,410	2,443	2,455

To return the pivot table from a split view to its original configuration, select any cell, and in the context menu select **Unsplit View** as illustrated in [Figure 27-23](#).

Figure 27-23 Pivot Table Unsplit View Command

		Acres					Electronvolts				
		Atlanta			Boston		Atlanta			Avoca	
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avoca
Aston Martin	Archie	2,449	2,482	2,492,982	3,015	3,027	2,107	2,140	2,152	2,	
	Betty	2,101	2,134	2,142,634	2,667	2,679	1,759	1,792	1,804	2,	
	Chet	1,819	1,852	1,862,352	2,385	2,397	1,477	1,510	1,522	2,	
	Daphne	2,483	2,516	2,523,016	3,049	3,061	2,141	2,174	2,186	2,	
	Ezra	2,083	2,116	2,122,616	2,649	2,661	1,741	1,774	1,786	2,	
Bentley	Archie	2,522	2,555	2,563,055	3,088	3,100	2,180	2,213	2,225	2,	
	Archie	2,836	2,869	2,883,369	3,402	3,414	2,494	2,527	2,539	3,	
Geo	Betty	2,488	2,521	2,533,021	3,054	3,066	2,146	2,179	2,191	2,	
	Chet	2,206	2,239	2,252,739	2,772	2,784	1,864	1,897	1,909	2,	
	Daphne	2,870	2,903	2,915,403	3,436	3,448	2,528	2,561	2,573	3,	
	Ezra	2,470	2,503	2,515,003	3,036	3,048	2,138	2,171	2,183	2,	
Harley-Davidson	Archie	2,831	2,864	2,876,364	3,397	3,409	2,489	2,522	2,534	3,	
	Betty	2,483	2,516	2,523,016	3,049	3,061	2,141	2,174	2,186	2,	
	Chet	2,201	2,234	2,242,734	2,767	2,779	1,859	1,892	1,904	2,	
	Daphne	2,865	2,898	2,910,398	3,431	3,443	2,523	2,556	2,568	3,	
Isdera	Ezra	2,465	2,498	2,510,998	3,031	3,043	2,123	2,156	2,168	2,	
	Archie	2,929	2,962	2,973,462	3,495	3,507	2,587	2,620	2,632	3,	
	Betty	2,581	2,614	2,626,114	3,147	3,159	2,239	2,272	2,284	2,	
	Chet	2,299	2,332	2,342,832	2,865	2,877	1,957	1,990	2,002	2,	
Daphne	Daphne	2,963	2,996	3,008,496	3,529	3,541	2,621	2,654	2,666	3,	
	Archie	2,045	2,078	2,090	1,755	1,788	1,800	2,410	2,443	2,455	

Sizing

The pivot table autosizes rows, columns, and layers within the space allowed when the pivot table is initially displayed. These rows, columns, and layers can be manually resized.

The default size of a pivot table is a width of 300 pixels and a height of 300 pixels. At runtime, you can change the size of rows, columns, or layers by dragging the row, column, or layer separator to a new location. Position the cursor in the row or column

header on the separator between the row, column, or layer you want to resize and the next row, column, or layer. When the cursor changes to a double-sided arrow, click and drag the row, column, or layer dotted line separator to the desired location. [Figure 27-24](#) shows the double-sided arrow and dotted line resize indicators.

Figure 27-24 Pivot Table Resize Indicators

US City	Cubits						Decibels										
	Avocados	Bobbins	Canoes														
Aston Martin	Archie	683	1,716	1,728	2,437	2,470	2,482	2,228	2,261	2,273	2,437	2,470	2,482	2,228	2,261	2,273	
	Betty	335	1,368	1,380	2,089	2,122	2,134	1,880	1,913	1,925	2,089	2,122	2,134	1,880	1,913	1,925	
	Chet	053	1,086	1,098	1,807	1,840	1,852	1,598	1,631	1,643	2,089	2,122	2,134	1,880	1,913	1,925	
	Daphne	717	1,750	1,762	2,471	2,504	2,516	2,262	2,295	2,307	1,807	1,840	1,852	1,598	1,631	1,643	
	Ezra	317	1,350	1,362	2,071	2,104	2,116	1,862	1,895	1,907	2,471	2,504	2,516	2,262	2,295	2,307	
Bentley	Archie	756	1,789	1,801	2,510	2,543	2,555	2,301	2,334	2,346	2,071	2,104	2,116	1,862	1,895	1,907	
	Bentley	Archie	756	1,789	1,801	2,510	2,543	2,555	2,301	2,334	2,346	2,071	2,104	2,116	1,862	1,895	1,907
	Betty	408	1,441	1,453	2,162	2,195	2,207	1,953	1,986	1,998							
	Chet	126	1,159	1,171	1,880	1,913	1,925	1,671	1,704	1,716							
	Daphne	790	1,823	1,835	2,544	2,577	2,589	2,335	2,368	2,380							
	Ezra	390	1,423	1,435	2,144	2,177	2,189	1,935	1,968	1,980							

When you resize rows, columns, or layers, the new sizes remain until you perform a pivot operation. After a pivot operation, the new sizes are cleared and the pivot table rows, columns, and layers return to their original sizes.

If you do not perform a pivot operation, then the new sizes remain for the life of the session. However, you cannot save these sizes through MDS (Metadata Services) customization.

Header Cell Word Wrapping

Pivot cells support word wrapping for header cells. By default, the text in header cell labels do not wrap if the text is longer than the default size of the header cell.

For long header labels you can set the `headerCell` component `whitespace` attribute to `normal` to enable word wrapping. The default value is `noWrap`. [Figure 27-10](#) shows a pivot table with row header cells wrapped to accommodate long text labels for Protective Gear and its drilled Black Hawk Knee Pads and Black Hawk Elbow Pads header cells.

Figure 27-25 Pivot Table Header Cell Word Wrapping

		101	111	109	107	106	112
		140	140	100	100		
▼	Gloves						
▶	Poles	925	25	630	520		280
▲	Protective Gear	1700			500		
	Black Hawk Knee Pads	850			500		
	Black Hawk Elbow Pads	850					
▶	Soccer	1325					
▶	Tires	3800	596				
▶	Weights	1900			250	820	
Average		772	190	298	257	388	78

Active Data Support (ADS)

Pivot tables and pivot filter bars support ADS by sending a Partial Page Refresh (PPR) request when an active data event is received.

The PPR response updates the pivot table and pivot filter bar values as follows:

- If the ADS event results in an update to the value of one or more existing pivot table data cells, the values are updated in place.
- If the ADS event results in an insert or delete of a row or column, or multiple rows or columns, the entire pivot table is refreshed to display the change.
- ADS is only supported for a single stamped `af:outputText` or `dvt:sparkChart` component in a data cell.
- If an event arrives while the pivot table is in an operation such as a pivot, the event is buffered so that it can be applied after the operation is completed; except in the case where the event is older than the data that the pivot operation just fetched, in which case the event is discarded.

For additional information about using the Active Data Service, see [Using the Active Data Service with an Asynchronous Backend](#).

Additional Functionality for the Pivot Table Component

You may find it helpful to understand other ADF Faces features before you implement your pivot table component. Additionally, once you have added a pivot table component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that pivot table components can use:

- You may want a pivot table to refresh a header cell, a data cell, or the entire pivot table to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).

- Personalization: If enabled, users can change the way the pivot table displays at runtime, and those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: Pivot table and pivot filter bar components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Skins and styles: You can customize the appearance of pivot table and pivot filter bar components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Content Delivery: You can configure your pivot table and pivot filter bar to fetch data from the data source immediately upon rendering the components, or on a second request after the components have been rendered using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound pivot tables based on how your ADF Business Components are configured. JDeveloper provides a wizard for data binding and configuring your pivot table. For more information, see the "Creating Databound Pivot Table and Pivot Filter Bar Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Pivot Table Component

To use the ADF DVT Pivot Table component, add the pivot table to a page using the Component Palette window. Then define the data for the pivot table and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Pivot Table Data Requirements

You can use any row set (flat file) data collection to supply data to a pivot table. The pivot table component uses a data model to display and interact with data.

The specific data model is `oracle.adf.view.faces.bi.model.DataModel`.

Pivot tables require that the `value` attribute is set in JDeveloper. If you are using UI-first development, the value of the `value` attribute must be stored in the pivot table's data model or in classes and managed beans.

Configuring Pivot Tables

The pivot table (`pivotTable`) component has two child components, a header cell (`headerCell`) and a data cell (`dataCell`). The pivot filter bar (`pivotFilterBar`) is a sibling component that can be associated with the pivot table. These components are defined by several configurable elements.

The prefix `dvt:` occurs at the beginning of each pivot table and pivot filter bar component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

Pivot tables display data in a grid layout with unlimited layers of hierarchically nested row header cells and column header cells. [Figure 27-26](#) shows a pivot table and its associated pivot filter bar displaying the sales of electronic equipment.

Figure 27-26 Electronic Sales Pivot Table

		Sales						
		Colorado	Oregon	Wyoming	Idaho	California	New Mexico	Utah
Audio Video	Ipod Nano 1Gb	1499.99	1499.99	1499.99				
	Ipod Nano 2Gb				199.95			
	Ipod Nano 3Gb					489.98		
	Ipod Shuffle 1Gb						99.99	
	Ipod Speakers							89.99
	Ipod Video 60Gb	399.99						399.99
	LCD HD Television							
	Tungsten E PDA	1999.99	1999.99	1999.99		1999.99	889.99	
Cell Phones	Plasma HD Telev...		199.95					
	Bluetooth Adaptor					225.99		
Product Total		3899.48	3695.4799999	3499.49	1099.94	2725.78	99.99	489.98

Pivot table and pivot filter bar components are defined by the following terms using the Electronic Sales Pivot Table in [Figure 27-26](#):

- Edges: The axes in pivot tables, including:
 - Row edge: The vertical axis to the left or right for right-to-left display of the body of the pivot table. In [Figure 27-26](#), the row edge contains two layers,

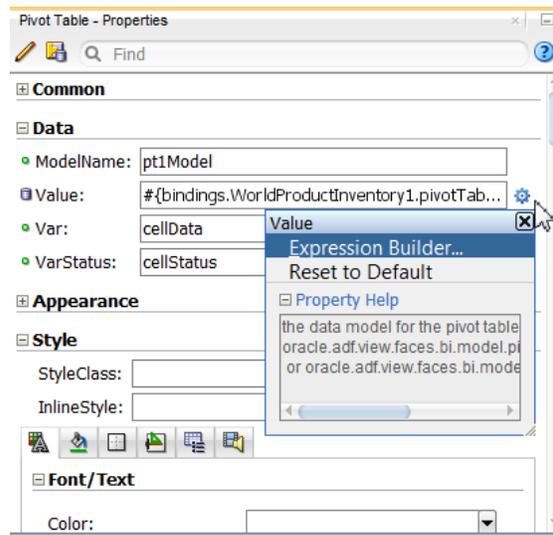
Product Category and Product, and each row in the pivot table represents the combination of a particular category and a particular product.

- Column edge: The horizontal axis above the body of the pivot table. In [Figure 27-26](#), the column edge contains two layers, Measure and US State, and each column in the pivot table represents the combination of a particular measure value (Sales or Units), and a particular geographic location (US State).
- Page edge: The edge represented by the pivot filter bar, whose layers can be filtered or pivoted with the layers in the row and column edges.
- Layers: Nested attributes that appear in a single edge. In [Figure 27-26](#), the following two layers appear in the column edge: Measure and Geography (Sales and US State). The following two layers appear in the row edge: Category and Product (Product Category and Product).
- Header cell: The labels that identify the data displayed in a row or column. Row header cells appear on the row edge, and column header cells appear on the column edge. In the sample, header cells include Cell Phones, iPod Speakers, Sales, and Colorado.
- Data cell: The cells within the pivot table that contain data values, not header information. In the sample, the first data cell contains a value of 1,499.99.
- QDR (Qualified Data Reference): A fully qualified data reference to a row, a column, or an individual cell. For example, in [Figure 27-26](#), the QDR for the first data cell in the pivot table must provide the following information:
 - Category=Audio Video
 - Product=iPod Nano 1Gb
 - Measure=Sales
 - Geography=ColoradoLikewise, the QDR for the first row in the pivot table, which is also the QDR of the "iPod Nano 1Gb" header cell, contains the following information:
 - Category=Audio Video
 - Product=iPod Nano 1GbFinally, the QDR for the "Sales" header cell contains the following information:
 - Measure=Sales

How to Add a Pivot Table to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a pivot table to the page. Once the pivot table is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the pivot table.

In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. [Figure 27-27](#) shows the dropdown menu for a pivot table component `value` attribute.

Figure 27-27 Pivot Table Value Attribute Dropdown Menu**Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a pivot table and the binding will be done for you. JDeveloper provides a wizard for data binding and configuring your pivot table. For more information, see the "Creating Databound Pivot Table and Pivot Filter Bar Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how pivot table attributes and child tags can affect functionality. For more information, see [Configuring Pivot Tables](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for the Pivot Table Component](#).

To add a pivot table to a page:

1. In the ADF Data Visualization page of the Components window, from the Pivot Table panel, drag and drop a **Pivot Table** onto the page.
2. In the Properties window, view the attributes for the pivot table. Use the help button to display the complete tag documentation for the `pivotTable` component.
3. Expand the **Data** section. Use this section to set the following attributes:
 - **Value:** Specify an EL expression for the object to which you want the pivot table to be bound. Can be an instance of `oracle.adf.view.faces.bi.model.DataModel`.
 - **Var** and **VarStatus:** Use to specify a variable to access cell data in stamped `dataCell` and `headerCell` components. For more information, see [Using var and varStatus Properties](#).

4. Expand the Appearance section. Use this section to set the following attributes:
 - **LayerLabelMode**: Use to configure the pivot table to always display the labels that appear above each row header layer and beside each column header layer. To configure persistent display of the row and column header labels for the `pivotTable` component, set the attribute to `rendered`. The default value is `hidden`.
 - **PivotLabelVisible**: Specify whether or not to display the labels on the pivot handles. The default value is `true`.
 - **PreferredColumnHeaderHeight** and **PreferredRowHeaderWidth**: As desired, use to specify the column header height and the row header width in percentages, for example 25% or 33.3%.
 - **Sizing**: Use to specify how the pivot table's size in width and height is determined. The default value is `fixed` where the pivot table is sized based on the parent component if the parent stretches its children, or the width and height CSS properties in its default skin, style class, or inline style if the parent does not stretch its children.

You can also set the attribute to `auto` where if the parent stretches its children, the pivot table is sized in the same way as the `fixed` attribute. If the parent does not stretch its children, the pivot table will shrink to the size of its content if the content is smaller than the pivot table, specifically:

`width=min(content width, default width)` and

`height=min(content height, default height)`

where "default width" and "default height" are the width and height CSS properties in its default skin, style class, or inline style.

 **Note:**

When this attribute is set to `auto`, the pivot table frame will initially be displayed with the default size of the pivot table and then readjusted to fit its contents. This can cause the layout of the page displaying the pivot table to change after the page is initially displayed.

- **StatusBarRendered**: Use to specify whether or not the pivot table status bar is display. The default value is `false`.
 - **EmptyText**: Enter the text to use to describe an empty pivot table. If the text is enclosed in an HTML tag, it will be formatted.
 - **Summary**: Enter a statement of the pivot table's purpose and structure for use by screen readers.
 - **DataFormat** and **HeaderFormat**: While a less preferred strategy to declaratively styling header and data cell stamps, you can use these attributes to create formatting rules to customize content in data and header cells. For more information, see [Formatting Pivot Table Cell Content With CellFormat](#).
5. Expand the **Behavior** section. Use this section to set the following attributes:
 - **PivotEnabled**: Specify whether or not to allow the end user to reposition the view of the data in the pivot table. The default value is `true`.

- **ColumnFetchSize** and **RowFetchSize**: Use to specify the number of columns and rows in a data fetch block. The default value for columns is 10 and the default value for rows is 25, which can be modified.
 - **ContentDelivery**: Use to specify how content will be delivered from the data source to the pivot table. The data can be delivered to the pivot table either by default as soon as the data is available (`whenAvailable`), immediately upon rendering (`immediate`), or lazily fetched after the shell of the component has been rendered (`lazy`). For more information about content delivery to pivot tables, see [Content Delivery](#).
6. In the Structure window, right-click the `dvt:pivotTable` node and choose **Insert Inside Pivot Table > Header Cell**.
 7. In the Structure window, right-click the `dvt:pivotTable` node and choose **Insert Inside Pivot Table > Data Cell**.

Configuring Pivot Table Display Size and Style

You can configure the pivot table, pivot filter bar, header cell and data cell's size and style using the `inlineStyle` or `styleClass` attributes.

Both attributes are available in the **Style** section in the Properties window for the `dvt:pivotTable`, `dvt:pivotFilterBar`, `dvt:headerCell`, or `dvt:dataCell` component. Using these attributes, you can customize stylistic features such as fonts, borders, and background elements.

You can also configure header cell and data cell child components using their styling attributes. For example, you can use custom CSS styling with `inlineStyle` and `contentStyle` attributes of a data cell `af:outputText` and `af:inputText` respectively:

```
<dvt:pivotTable id="goodPT"
    value="#{richPivotTableModel.dataModel}"
    var="cellData"
    varStatus="cellStatus">
  <dvt:dataCell id="dc1">
    <af:switcher id="sw1" facetName="#{richPivotTableModel.stampFacet}">
      <f:facet name="outputText">
        <af:outputText id="ot1" value="#{cellData.dataValue}"
            inlineStyle="#{myBean.textStyle}" />
      </f:facet>
      <f:facet name="inputText">
        <af:inputText id="ot2" value="#{cellData.dataValue}"
            contentStyle="#{myBean.textStyle}" />
      </f:facet>
    </af:switcher>
  </dvt:dataCell>
</dvt:pivotTable>
```

Pivot tables and pivot filter bars also support skinning to customize many aspects of the display of data and header cells and labels, and pivoting and sorting icons.

For the complete list of pivot table skinning keys, see the *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces Skin Selectors*. For additional information about customizing your application using skinning and styles, see [Customizing the Appearance Using Styles and Skins](#).

The page containing the pivot table may also impose limitations on the ability to change the size or style. For more information about page layouts, see [Organizing Content on Web Pages](#).

What Happens When You Add a Pivot Table to a Page

When a pivot table component is inserted into a JSF page using the Components window, a basic pivot table tag is added to the source code as follows:

```
<dvt:pivotTable id="pt1"/>
```

If you have not already done so, you can then use the Components window to insert a header cell and data cell. Configure the cell content through stamping. For more information, see [Configuring Header and Data Cell Stamps](#).

A Create Pivot Table wizard provides declarative support for data-binding and configuring the pivot table. In the wizard pages you can:

- Specify the initial layout of the pivot table
- Associate and configure a pivot filter bar
- Specify alternative labels for the data layers
- Configure insert or filter drilling
- Define aggregation of data values
- Configure category and data sorting

For more information, see the "Creating Databound Pivot Tables" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

What You May Need to Know About Displaying Large Data Sets

When you are developing an ADF Faces web application, by default pivot tables use a vertical or horizontal scroll bar for displaying rows over the size of the data being fetched. Alternatively, you can configure a vertical or horizontal page control that allows users to jump to specific pages of rows as illustrated in [Figure 27-19](#). To configure a page control for desktop devices, set the `pivotTable` component `scrollPolicy` attribute to `page`.

By default, when rendered on mobile devices, pivot tables use a page control for displaying rows over the size of the data being fetched. For pivot tables to display on a mobile device, you must:

- Place the pivot table component within a flowing container (that is, a component that does not stretch its children). For more information about flowing container components, see [Geometry Management and Component Stretching](#).
- Leave the `scrollPolicy` attribute set to `auto` (default for this setting on mobile devices is paginated display of the pivot table).

If the pivot table is not in a flowing container, or if those attributes are not set correctly, the pivot table will display a scroll bar instead of pages.

 **Note:**

For row-based data sets with more than 1000 rows, pre-calculate the summary data in the database to limit the number of rows sent to the pivot table, and thereby reduce resources necessary for data aggregation. You can tune your view object to set how data is retrieved from the database. Use the default setting of **All Rows** in batches of 1. For more information, see "Consider the Appropriate Tuning Setting for Every View Object" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

You can also specify the number of columns and rows in a data fetch block by setting the `columnFetchSize` and `rowFetchSize` attributes of the `dvt:pivotTable` component.

What You May Need to Know About Pivot Tables on Touch Devices

The ADF Faces framework is optimized to run in mobile browsers such as Safari. The framework recognizes when a mobile browser on a touch device is requesting a page, and then delivers only the JavaScript and peer code applicable to a mobile device. However, while a standard ADF Faces web application will run in mobile browsers, because the user interaction is different and because screen size is limited, when your application needs to run in a mobile browser, you should create touch device-specific versions of the pages. For more information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).

What You May Need to Know About Skinning and Customizing the Appearance of Pivot Tables

For the complete list of pivot table skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Configuring Header and Data Cell Stamps

Each immediate child of a ADF DVT Pivot Table component must be either a `headerCell` or `dataCell` component. The pivot table can contain at most one `headerCell` and at most one `dataCell` component. These components make it possible to customize the cell content through stamping.

When you use stamping, child components are not created for every header cell or data cell in a pivot table. Rather, the content of the component is repeatedly rendered, or stamped, once per cell.

Each time a header or data cell is stamped, the value for the current cell is copied into a `var` property, and additional data for the cell is copied into a `varStatus` property. These properties can be accessed in EL expressions inside the header or data cell component, for example, to pass the cell value to a stamped `af:outputText`

component. Once the pivot table has completed rendering, the `var` and `varStatus` properties are removed, or reverted back to their previous values.

Using `var` and `varStatus` Properties

Pivot table `var` and `varStatus` properties are used to access cell data in stamped `dataCell` and `headerCell` components. The `var` property names the EL expression variable used to reference cell data within pivot table data cell stamps. In the stamped `dataCell` or `headerCell` component, the `var` property must be referenced and followed by a metadata keyword.

[Table 27-1](#) shows the metadata keywords supported for data cells in a rowset data model.

Table 27-1 Supported Metadata Keywords for Data Cells

Keyword	Description
<code>dataValue</code>	Most frequently useful keyword. Returns the data value <code>Object</code> for the current cell. To specify the object's accessible field through EL Expression, use the setting <code>dataValue.fieldName</code> .
<code>dataCubeMax</code> and <code>dataCubeMin</code>	Returns a number that is the maximum and minimum, respectively, for the measure of the cell across the values in the cube.
<code>dataIsTotal</code>	Returns a Boolean <code>true</code> if this cell is an aggregate.
<code>dataAggregates</code>	If the cell is an aggregate, returns a <code>List<String, Object></code> of the column and value pairs representing the cells (nonaggregate) that make up the aggregation for the given cell.
<code>aggregateCollection</code>	If the cell is an aggregate, returns the <code>List<String, Object></code> of the column and value pairs in the cube that make up the cell's aggregate value. Note that <code>aggregateCollection</code> is post-cube and <code>dataAggregates</code> is not.
<code>dataRow</code>	Returns a <code>Map<String, Object></code> from attribute name to data <code>Object</code> in the original row mapping. Usage: <code>dataRow.foo</code> , where "foo" is one of the rowset attribute (column) names.
<code>dataTypeColumn</code>	Returns a <code>String</code> representing the name of the rowset attribute from which the value comes.
<code>dataRowKey</code>	Returns the row data model's ADF Model row key,
<code>dataKeyPath</code>	Returns the ADF Model key path object.

[Table 27-2](#) shows the metadata keywords supported for header cells in a rowset data model.

Table 27-2 Supported Metadata Keywords for Header Cells

Keyword	Description
<code>dataValue</code>	Most frequently useful keyword. Returns the data value <code>Object</code> for the current cell. To specify the object's accessible field through EL Expression, use the setting <code>dataValue.fieldName</code> .
<code>value</code>	Returns the <code>String</code> value of the header cell. Also available in cubic data models.

Table 27-2 (Cont.) Supported Metadata Keywords for Header Cells

Keyword	Description
label	Returns the String label for the header cell. Also available in cubic data models.
isTotal	Returns a Boolean <code>true</code> if the header cell represents an aggregate.
drillState	Returns an Integer value representing the drill state of the current header cell, if applicable. 0 indicates "not drillable", 1 indicates "drillable", and 2 indicates "drilled". Also available in cubic data models
memberMetadataColumn	Returns the String attribute column of the header cell.
layerName	Returns a String representing the name of the layer containing the header cell.
layerLabel	Returns a String representing the label (if any) for the layer containing this header cell. May fall back to <code>layerName</code> .

The optional `varStatus` property names the EL expression variable used to provide contextual information about the state of the component. In stamped `dataCell` or `headerCell` components, the `varStatus` property must be referenced and followed by one of the following:

- `members`: Valid only for the `dataCell` component. Provides access to the header cells corresponding to the same row or column as the current data cell.
- `model`: Returns the `DataModel` for this component.
- `cellIndex`: Returns the cell index for this component.
- `cellKey`: Returns the cell key for this component.

For example you can use `var` and `varStatus` to access data from a stamped data cell and to format the pivot table based on the header cell stamp:

```
<dvt:pivotTable
    id="pivotTable3"
    value="#{pivotTableMemberFormatting.dataModel}"
    var="cellData"
    varStatus="cellStatus">
  <dvt:headerCell>
    <af:switcher
      facetname="0__b_cellData_layerName__b__"
      defaultFacet="Other">
      <f:facet name="Product">
        <af:outputText id="ot1"
          value="#{cellData.dataValue}"
          inlineStyle="color:#{(cellData.dataValue == 'Canoes' ?
            'red' : 'blue')}"/>
      </f:facet>
      <f:facet name="Other">
        <af:outputText id="ot2" value="#{cellData.dataValue}"/>
      </f:facet>
    </af:switcher>
  </dvt:headerCell>

  <dvt:dataCell>
```

```

<af:outputText id="ot3" value="#{cellData.dataValue}"
  inlineStyle="color:#{(cellStatus.members.Product.dataValue ==
  'Canoes' ? 'red' : 'blue')};"/>
</dvt:dataCell>
</dvt:pivotTable>

```

The code sample illustrates the syntax for using each data cell value property as follows:

- `var`: [var property].[data cell metadata keyword]
In the code sample, the value of `af:outputText` is set to `#{cellData.dataValue}`, the value of the current cell.
- `varStatus`: [varStatus property].[members].[layer name].[header cell metadata keyword]
The data cell component value references the pivot table `varStatus` (`cellStatus`) followed by `members` to access the header cells corresponding to the same row or column as the current data cell, followed by the name of the layer (`Product`) containing the desired header cell, followed by the header cell metadata keyword `dataValue`.

[Using var and varStatus Properties](#) shows the pivot table resulting from the code sample.

Figure 27-28 Pivot Table with Formatting Based Header Cell Stamp

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20000.0	500.0	200.0	50.0
	Canoes	15000.0	1500.0	75.0	8.0
2006	Tents	10000.0	250.0	100.0	25.0
	Canoes	7500.0	750.0	40.0	4.0
2005	Tents	5000.0	125.0	50.0	15.0
	Canoes	3750.0	375.0	20.0	2.0

You can also use `var` and `varStatus` to stamp sparkcharts in pivot tables: and gauges in pivot tables:

```

<dvt:pivotTable id="pivotTable1"
  value="#{pivotTableSparkChart.dataModel}"
  var="cellData"
  varStatus="cellStatus">
  <dvt:dataCell>
  <af:switcher id="s2"
    facetname="0__b_cellData_dataIsTotal__b__"
    defaultFacet="false">
    <f:facet name="true">
    <dvt:sparkChart id="scl" shortDesc="Spark Chart"
      highMarkerColor="#008200"
      lowMarkerColor="#ff0000">
    <af:iterator id="i1"
      value="#{cellData.aggregateCollection}"
      var="sparks" >
    <dvt:sparkItem id="sil"
      value="#{sparks.dataValue}"/>
    </af:iterator>
    </dvt:sparkChart>
    </f:facet>

```

```

        <f:facet name="false">
            <af:outputText id="ot1" value="#{cellData.dataValue}"/>
        </f:facet>
    </af:switcher>
</dvt:dataCell>

<dvt:headerCell>
    <af:switcher id="s3"
        facetname="0___b_cellData_isTotal___b_"
        defaultFacet="false">
        <f:facet name="true">
            <af:outputText id="ot2" value="Trend"/>
        </f:facet>
        <f:facet name="false">
            <af:outputText id="ot3" value="#{cellData.dataValue}"/>
        </f:facet>
    </af:switcher>
</dvt:headerCell>
</dvt:pivotTable>

```

The resulting pivot table is shown in [Figure 27-2](#).

You can also use `var` and `varStatus` to stamp gauges in pivot tables:

```

<dvt:pivotTable
    id="pivotTable2"
    value="#{pivotTableGauge.dataModel}"
    var="cellData"
    varStatus="cellStatus">
    <dvt:dataCell>
        <dvt:gauge id="g1" shortDesc="Gauge"
            imageWidth="80" imageHeight="80" imageFormat="PNG_STAMPED"
            value="#{cellData.dataValue}"
            minValue="#{cellData.dataCubeMin}"
            maxValue="#{cellData.dataCubeMax}"/>
    </dvt:dataCell>
</dvt:pivotTable>

```

The resulting pivot table is displayed in [Figure 27-3](#).

How to Configure Header and Data Cell Stamps

Only certain types of child components are supported by header cells or data cells. For example, each header cell can contain read-only components. Each data cell can contain read-only or input components, including all components with no activity and most components that implement the `EditableValueHolder` or `ActionSource` interfaces.

Header cells and data cells should have only one child component. If multiple children are desired, they should be wrapped in another component. If no layout is desired, `af:group` can be used, which simply renders its children without adding layout, and is consequently lightweight. If layout is desired, a layout component like `af:panelGroupLayout` can be used instead. For more information, see [Grouping Related Items](#).

Data cell editing is enabled by using an input component as the child component of `dataCell`. At runtime you can open the cell for editing by clicking the cell in the pivot table. For more information, see [Editing Data Cells](#).

You can configure header cell stamping using `af:switcher` to vary the type of stamped component by layer name, that is, a different content for Geography, Channel, and so on using components as children of `headerCell`:

```
<dvt:pivotTable id="goodPT"
    inlineStyle="width:100%;height:600px;"
    binding="#{editor.component}"
    contentDelivery="immediate"
    value="#{pivotTableHeaderCellDemo.dataModel}"
    headerFormat="#{pivotTableHeaderCellDemo.getHeaderFormat}"
    dataFormat="#{pivotTableHeaderCellDemo.getDataFormat}"
    var="cellData"
    varStatus="cellStatus"
    summary="pivot table">
<dvt:headerCell id="goodHC">
  <af:switcher id="sw" facetName="#{cellData.layerName}" defaultFacet="Other">
    <f:facet name="Geography">
      <af:group id="g1">
        <af:icon id="idicon11" name="info" shortDesc="Icon" />
        <af:outputText value="#{cellData.dataValue}" id="ot11"
            shortDesc="#{cellData.dataValue}" />
      </af:group>
    </f:facet>
    <f:facet name="Channel">
      <af:group id="g2">
        <af:panelGroupLayout id="pgl2" layout="vertical">
          <af:link shortDesc="Sample Link"
              icon="/images/pivotTableCSVDemo/smily-normal.gif"
              hoverIcon="/images/pivotTableCSVDemo/smily-glasses.gif"
              id="cill1"/>
          <af:outputText value="#{cellData.dataValue}" id="ot1" />
        </af:group>
        <af:button text="Go to Tag Guide page" immediate="true"
            action="guide" id="cb1"/>
      </af:panelGroupLayout>
    </f:facet>
    <f:facet name="Product">
      <af:panelGroupLayout id="pgl3" layout="vertical">
        <af:outputText value="#{cellData.dataValue}" id="ot12" />
        <af:button text="Go to Tag Guide page" immediate="true"
            action="guide" id="cb2"/>
      </af:panelGroupLayout>
    </f:facet>
    <f:facet name="Other">
      <af:link text="#{cellData.dataValue}"
          shortDesc="#{cellData.dataValue}" immediate="true"
          action="guide" id="idlink11"/>
    </f:facet>
  </af:switcher>
</dvt:headerCell>
</dvt:pivotTable>
```

Figure 27-29 shows the resulting pivot table for the code sample.

Figure 27-29 Pivot Table Header Cell Stamps

		Sales		Units	
		☺ All Channels		☺ All Channels	
		Go to Tag Guide page		Go to Tag Guide page	
		World	Boston	World	Boston
2007	Tents Go to Tag Guide page	20,000.000	500.000	200.000	50.000
	Canoes Go to Tag Guide page	15,000.000	1,500.000	75.000	8.000
2006	Tents Go to Tag Guide page	10,000.000	250.000	100.000	25.000
	Canoes Go to Tag Guide page	7,500.000	750.000	40.000	4.000
2005	Tents Go to Tag Guide page	5,000.000	125.000	50.000	15.000
	Canoes Go to Tag Guide page	3,750.000	375.000	20.000	2.000

You can configure data cell stamping using `af:switcher` to vary the type of stamped component by measure, that is, a different content for Sales, Weight, and so on using components as children of `dataCell`:

```
<dvt:pivotTable id="goodPT" var="cellData" varStatus="cellStatus">
  <dvt:dataCell>
    <af:switcher id="sw" facetName="#{cellStatus.members.MeasDim.value}"
      defaultFacet="Other">
      <f:facet name="Sales">
        <af:inputText id="idinputtext1" value="#{cellData.dataValue}" />
      </f:facet>
      <f:facet name="Units">
        <af:inputText id="idinputtext2" value="#{cellData.dataValue}" />
        <af:validateLength maximum="6" minimum="2" />
      </f:facet>
      <f:facet name="Weight">
        <af:outputText id="idoutputtext1" value="#{cellData.dataValue}" />
      </f:facet>
      <f:facet name="Color">
        <af:selectOneChoice id="idselectonechoice"
          value="#{cellData.dataValue}" label="Color">
          <af:selectItem label="red" value="red" shortDesc="shortDesc sample"/>
          <af:selectItem label="coffee" value="coffee"
            shortDesc="Sample shortDesc text"/>
          <af:selectItem label="milk" value="milk"
            shortDesc="Another shortDesc sample"/>
        </af:selectOneChoice>
      </f:facet>
      <f:facet name="Available">
        <af:selectBooleanCheckbox id="idselectbooleancheckbox"
          label="Availability" text="Item Available"
          autoSubmit="true"
          value="#{cellData.dataValue}" />
      </f:facet>
      <f:facet name="Supply Date">
        <af:inputDate id="idinputdate1" value="#{cellData.dataValue}"
          label="Change Date:" simple="true" />
        <af:validateDateTimeRange maximum="2020-12-31" minimum="1980-12-31" />
      </af:inputDate>
      </f:facet>
    </af:switcher>
  </dvt:dataCell>
</dvt:pivotTable>
```

```

<f:facet name="Link">
  <af:link text="#{cellData.dataValue}" immediate="true"
          action="guide" id="idlink"/>
</f:facet>
<f:facet name="Size">
  <af:inputComboboxListOfValues label="Size" id="idInputComboboxListOfValues"
                               value="#{cellData.dataValue}"
                               searchDesc="Search Size"
                               model="#{pivotTableEditBean.listOfValuesModel}"
                               columns="3" />
</f:facet>
<f:facet name="Other">
  <af:outputText id="idoutputtext2" value="#{cellData.dataValue}" />
</f:facet>
</af:switcher>
</dvt:dataCell>
</dvt:pivotTable>

```

Figure 27-30 shows the resulting pivot table for the code sample.

Figure 27-30 Pivot Table Data Cell Stamps

				Sales	Units	Available	Price	Color	Weight	Link	Size	Supply Date
2007	Tents	All Channels	World	20000.0	200.0	✓ Item Available	33.0	red	33.0	Main-link	L	1/1/2000
			Boston	500.0	50.0	— Item Available	66.0	coffee	66.0	Main-link	S	4/4/2004
	Jacket	All Channels	World	40000.0	4000.0	✓ Item Available	11.0	red	44.0	Main-link	L	1/1/2000
			Boston	700.0	70.0	— Item Available	13.0	coffee	42.0	Main-link	S	4/4/2004
2008	Tents	All Channels	World	30000.0	300.0	✓ Item Available	36.0	red	33.0	Main-link	L	1/1/2000
			Boston	3000.0	30.0	— Item Available	69.0	coffee	66.0	Main-link	S	4/4/2004
	Jacket	All Channels	World	10000.0	1000.0	✓ Item Available	12.0	red	44.0	Sub-link	M	3/3/2003
			Boston	600.0	60.0	— Item Available	15.0	coffee	42.0	Main-link	S	4/4/2004

Before you begin:

It may be helpful to have an understanding of how pivot table attributes and child tags can affect functionality. For more information, see [Configuring Pivot Tables](#).

You should already have a pivot table on your page. If you do not, follow the instructions in this chapter to create a pivot table. For more information, see [How to Add a Pivot Table to a Page](#).

To add and configure a header or data cell stamp:

1. In ADF Data Visualization page of the Components window, from the Pivot Table panel, drag and drop a **Header Cell** or **Data Cell** onto the pivot table in the visual editor.
2. In the Structure window, right-click the **dvt:headerCell** or **dvt:dataCell** and choose **insert inside Header Cell** or **insert inside Data Cell > ADF Data Visualization Components** or **ADF Faces**.
3. In the Insert Item dialog, select the component you wish to stamp in the header or data cell.
4. In the Structure window, select the component you inserted, and in the Properties window, set the component attributes.

Using Pivot Filter Bars

You can enhance the data filtering capacity in a pivot table by adding an ADF DVT Pivot Filter Bar. Any data layers specified in the pivot table can be moved to or specified in the filter bar to filter the displayed data.

Zero or more layers of data not already displayed in the pivot table row edge or column edge are displayed in the page edge. [Figure 27-31](#) shows a pivot filter bar with Quarter and Month layers that can be used to filter the data displayed in the pivot table.

Figure 27-31 Pivot Filter Bar with Data Layer Filters

		Massachusetts	Rhode Island
Audio Components	Direct	5,000	2,000
	Indirect	500	200
Video Components	Direct	10,000	500
	Indirect	1,000	50
Gaming Systems	Direct	10,000	700
	Indirect	1,000	70
Photography Equipment	Direct	7,000	3,050
	Indirect	700	305
Phones	Direct	10,500	1,050
	Indirect	1,050	105
Miscellaneous	Direct	10,500	1,050
	Indirect	1,000	105

You can also change the display of data in the pivot table by pivoting layers between the row, column, or page edges. Use the pivot handle to drag the layers between the edges as desired. [Figure 27-32](#) shows the modified pivot table and pivot filter bar when the Channel data layer is pivoted to the page edge.

Figure 27-32 Pivot Table and Pivot Filter Bar After Pivot

	Massachusetts	Rhode Island
Audio Components	5,000	2,000
Video Components	10,000	500
Gaming Systems	10,000	700
Photography Equipment	7,000	3,050
Phones	10,500	1,050
Miscellaneous	10,500	1,050

You can style pivot filter bars using `inlineStyle` and `styleClass` attributes and skinning keys. See [Configuring Pivot Table Display Size and Style](#).

Using a Pivot Filter Bar with a Pivot Table

You can use a pivot filter bar component, `pivotFilterBar`, to work with a pivot table component, `pivotTable`, by configuring the data model and associated properties to work with both components:

```
<dvt:pivotFilterBar id="pf1" value="#{binding.pt.pivotFilterBarModel}"
  modelName="pt1Model"/>
<dvt:pivotTable id="pt1" value="#{binding.pt.dataModel}" modelName="pt1Model"
  partialTriggers="pf1"/>
```

You can associate a pivot filter bar with a pivot table in any of the following ways:

- Create a pivot table using the Data Controls Panel.

When you drag a data collection from the Data Controls Panel to create a pivot table on your page, the Select Display Attributes page of the Create Pivot Table wizard provides the option to create a pivot filter bar to associate with the pivot table. You can choose to specify zero or more attributes representing data layers in the page edge. The data model and associated properties are automatically configured for you. For detailed information, see the "Creating Databound Pivot Tables" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Add a pivot filter bar to a pivot table bound to data.

In the ADF Data Visualizations page of the Components window, from the Pivot Table panel, you can drag a `pivotFilterBar` element adjacent to a `pivotTable` element that has been bound to a data collection and the data binding will be done for you.

- Add a pivot filter bar to a pivot table not bound to data.

In the ADF Data Visualizations page of the Components window, from the Pivot Table panel, you can drag a `pivotFilterBar` element adjacent to a `pivotTable` element that has not been bound to a data collection. In this instance, you must configure the data model and associated properties in order for the pivot filter bar to work with the pivot table.

Using a Pivot Filter Bar with a Graph

You can use a pivot filter bar to filter the graphical display of data in a graph. For example, you can show a filtered view of quarterly sales data displayed in both a pivot table and on a graph as illustrated in [Figure 27-6](#).

Use partial page rendering (PPR) to configure the pivot filter bar as a trigger with a pivot table and a graph as targets. Once PPR is triggered, any component configured to be a target will be rerendered. You configure a component to be a target by setting the `partialTriggers` attribute to the relative ID of the trigger component. For information about relative IDs, see [Locating a Client Component on a Page](#). For more information about PPR, see [Rerendering Partial Page Content](#).

For example, to configure the pivot table and graph in [Figure 27-6](#):

```
<dvt:pivotFilterBar id="pfb1" binding="#{editor.component}"
  value="#{pivotFilterBar.queryDescriptor}" modelName="model1"
  styleClass="AFStretchWidth"/>
<af:panelGroupLayout layout="horizontal" id="pg12">
```

```

<f:facet name="separator" >
  <af:separator id="s2"/>
</f:facet>
<af:spacer width="25px" id="s3"/>
  <dvt:pivotTable id="pt1" inlineStyle="width:400px" partialTriggers="::pfb1"
    value="#{pivotFilterBar.dataModel}" modelName="model1"
    summary="Quarterly Sales Pivot Table"/>
<af:spacer width="50px" id="s4"/>
  <dvt:barGraph id="bar1" partialTriggers="::pfb1 ::pt1"
    value="#{pivotFilterBar.dataModel}" shortDesc="Quarterly Sales
    Bar Graph"/>
</af:panelGroupLayout>

```

What You May Need to Know About Skinning and Customizing the Appearance of Pivot Filter Bars

For the complete list of pivot filter bar skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactivity to Pivot Tables

ADF DVT Pivot tables and pivot filter bars support multiple user operations, including data selection, exporting to an spreadsheet, and displaying in printable mode.

Using Selection in Pivot Tables

Selection in a pivot table allows a user to select one or more cells in a pivot table. Only one of the three areas including the row header, column header, or data cells can be selected at one time.

An application can implement features such as displaying customized content for a context menu, based on currently selected cells. For example, to get the currently selected header cells:

```

UIPivotTable pt = getPivotTable()
if (pt == null)
  return null;
HeaderCellSelectionSet headerCells = null;
if (pt.getSelection().getColumnHeaderCells().size() > 0) {
  headerCells = pt.getSelection().getColumnHeaderCells();
} else if (pt.getSelection().getRowHeaderCells().size() > 0) {
  headerCells = pt.getSelection().getRowHeaderCells();
}

```

At runtime, selecting a data cell highlights the cell, as shown in [Figure 27-11](#).

Figure 27-33 Selected Data Cell

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20,000	500	200	50
	Canoes	15,000	1,500	75	8
2006	Tents	10,000	250	100	25
	Canoes	7,500	750	40	4
2005	Tents	5,000	125	50	15
	Canoes	3,750	375	20	2

Using Partial Page Rendering

You can update pivot tables, data cells, and header cells by using partial page rendering (PPR). For example, you may display totals in a pivot table when triggered by a checkbox. PPR allows individual components on a page to be rerendered without the need to refresh the entire page. For more information about PPR, see [About Partial Page Rendering](#).



Note:

By default, ADF pivot tables support automatic PPR, where any component whose values change as a result of backend business logic is automatically rerendered. If your application uses the Fusion technology stack, you can enable the automatic partial page rendering feature on any page. For more information, see the "What You May Need to Know About Partial Page Rendering and Iterator Bindings" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

For a component to be rerendered based on an event caused by another component, it must declare which other components are the triggers. Use the `partialTriggers` attribute to provide a list of IDs of the components that should trigger a partial update of the pivot table. The pivot table listens on the trigger components and if one of the trigger components receives an event that will cause it to update in some way, the pivot table is also updated.

For example, you can update a pivot table by displaying the totals when a checkbox is triggered using the triggering component ID as the `partialTriggers` value.

```
<dvt:pivotTable id="goodPT"
  value="#{richPivotTableModel.dataModel}"
  partialTriggers="showTotals"/>

<af:selectBooleanCheckbox id="showTotals" autoSubmit="true" label="Show Totals"
  value="#{richPivotTableModel.totalsEnabled}"/>
```

Exporting from a Pivot Table

You can export the data from a pivot table to a Microsoft Excel spreadsheet. You create an action source, such as a button or link, add a `exportPivotTableData` component, and associate it with the data you wish to export. You can configure the component so that the entire pivot table will be exported, or so that only the rows, columns, or data cells selected by the user will be exported. For example,

Figure 27-34 shows a pivot table that includes button components that allow users to export the data to an Excel spreadsheet.

Figure 27-34 Pivot Table with Export to Excel Buttons

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20,000	500	200	50
	Canoes	15,000	1,500	75	8
2006	Tents	10,000	250	100	25
	Canoes	7,500	750	40	4
2005	Tents	5,000	125	50	15
	Canoes	3,750	375	20	2

Export All Export Selected

At runtime, when the user clicks the button, by default all the rows and columns are exported in an Excel format written to the file specified in the `filename` attribute of the component. Alternatively, you can configure the `exportPivotTableData` component so that only user selections are exported, by setting the `exportedData` attribute to `selected`. For example, for the **Export to Excel** buttons:

```
<dvt:pivotTable id="pivotTableToExport"
    binding="#{editor.component}"
    contentDelivery="immediate"
    value="#{pivotTableExport.dataModel}" summary="pivot table"/>

<h:panelGrid id="pf1" columns="2" cellpadding="3">
  <af:button text="Export All" id="exportAll">
    <dvt:exportPivotTableData exportedId="pivotTableToExport" type="excelHTML"
      exportedData="all" filename="all.xls"
      title="All pivotTable data"/>
  </af:button>
  <af:button text="Export Selected" id="exportSelected">
    <dvt:exportPivotTableData exportedId="pivotTableToExport" type="excelHTML"
      exportedData="selected" filename="selected.xls"
      title="Selected pivotTable data"/>
  </af:button>
</h:panelGrid>
```

Figure 27-35 shows the resulting Excel spreadsheet when the Export All button is clicked.

Figure 27-35 Pivot Table Export to Excel Spreadsheet

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - updated_export.xls". The spreadsheet contains the following data:

		Sales		Units	
		All Channels		All Channels	
		World	Boston	World	Boston
2007	Tents	20,000.000	500.000	200.000	50.000
	Canoes	15,000.000	1,500.000	75.000	8.000
2006	Tents	10,000.000	250.000	100.000	25.000
	Canoes	7,500.000	750.000	40.000	4.000
2005	Tents	5,000.000	125.000	50.000	15.000
	Canoes	3,750.000	350.000	20.000	2.000

**Note:**

You may receive a warning from Excel stating that the file is in a different format than specified by the file extension. This warning can be safely ignored.

Displaying Pivot Tables in Printable Pages

ADF Faces allows you to output your JSF page from an ADF Faces web application in a simplified mode for printing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scrollbars and buttons. For information about creating simplified pages for these outputs, see [Using Different Output Modes](#).

When a pivot table and pivot filter bar is displayed on a JSF page to be output in printable pages:

- All data cells in the pivot table are displayed.
- Limited client interactivity including cell select and row or column resizing is supported.
- Pivoting, drilling, and sorting operations are not supported.
- Context menus including the ability to resize rows or columns is not supported.
- If configured, the pivot table data filter displayed in the pivot filter bar will be displayed, although the contents cannot be changed.

Formatting Pivot Table Cell Content With CellFormat

Although a less preferred strategy, you can use a `CellFormat` method expression as an alternative to declaratively styling header and data cell stamps in ADF DVT Pivot Tables.

For more information about using `inlineStyle` and `styleClass` attributes. See [Configuring Pivot Table Display Size and Style](#).

All cells in a pivot table are either header cells or data cells. Before rendering a cell, the pivot table calls a method expression. You can customize the content of pivot table header cells and data cells by providing method expressions for the following attributes of the `pivotTable` component:

- For header cells, use one of the following attributes:
 - `headerFormat`: Use to create formatting rules to customize header cell content.
 - `headerFormatManager`: Use only if you want to provide custom state saving for the formatting rules of the application's pivot table header cells.
- For data cells, use one of the following attributes:
 - `dataFormat`: Use to create formatting rules to customize data cell content.
 - `dataFormatManager`: Use only if you want to provide custom state saving for the formatting rules of the application's pivot table data cells.

Using a CellFormat Object for a Data Cell

To specify customization of the content of a data cell, you must code a method expression that returns an instance of `oracle.dss.adf.view.faces.bi.component.pivotTable.CellFormat`.

An instance of a `CellFormat` object lets you specify an argument to change the CSS style of a cell. For example, you might use this argument to change the background color of a cell.

- **Converter:** An instance of `javax.faces.convert.Converter`, which is used to perform number, date, or text formatting of a raw value in a cell.
- **CSS style:** Used to change the CSS style of a cell. For example, you might use this argument to change the background color of a cell.
- **CSS text style:** Used to change the CSS style of the text in a cell. For example, you might use this argument to set text to bold.
- **New raw value:** Used to change the cell's underlying value that was returned from the data model. For example, you might choose to change the abbreviated names of states to longer names. In this case, the abbreviation NY might be changed to New York.

To create an instance of a `CellFormat` object for a data cell:

1. Construct an `oracle.adf.view.faces.bi.component.pivotTable.DataCellContext` object for the data cells that you want to format. The `DataCellContext` method requires the following parameters in its constructor:
 - `model`: The name of the `dataModel` used by the pivot table.
 - `row`: An integer that specifies the zero-based row that contains the data cell on which you are operating.
 - `column`: An integer that specifies the zero-based column that contains the data cell that you want to format.
 - `qdr`: The `QDR` that is a fully qualified reference for the data cell that you want to format.
 - `value`: A `java.lang.Object` that contains the value in the data cell that you want to format.
2. Pass the `DataCellContext` to a method expression for the `dataFormat` attribute of the pivot table.
3. In the method expression, write code that specifies the kind of formatting you want to apply to the data cells of the pivot table. This method expression must return a `CellFormat` object.

Specifying a Cell Format

You can apply header and data cell formatting styles to emphasize aspects of the data displayed in the pivot table. [Figure 27-36](#) shows a pivot table with sales totals generated for products and for product categories. In the rows that contain totals, this pivot table displays text against a shaded background, a style change. This change

shows in both the row header cells and the data cells for the pivot table. The row headers for totals contain the text "Sales Total."

The pivot table also shows stoplight and conditional formatting of data cells. For more information, see [Configuring Stoplight and Conditional Formatting Using CellFormat](#) .

Figure 27-36 Sales Data Per Product Category

Sales Data Per Product Category		Colorado	Oregon	Wyoming	Idaho	California	N
Audio and Video	Ipod Nano 1Gb	\$1,499.50	\$1,499.50	\$1,499.50			
	Ipod Nano 2Gb				\$199.95		
	Ipod Nano 4Gb					\$499.90	
	Ipod Shuffle 1Gb						
	Ipod Speakers						
	Ipod Video 60Gb	\$399.99					
	LCD HD Television				\$899.99		
	Plasma HD Television	\$1,999.99	\$1,999.99	\$1,999.99		\$1,999.99	
	Tungsten E PDA		\$195.99				
	Zune 30Gb					\$225.99	
Sales Total	\$3,899.48	\$3,695.48	\$3,499.49	\$1,099.94	\$2,725.88		
Cell Phones	Bluetooth Adaptor						
	Bluetooth Headset	\$49.99		\$49.99	\$149.97		
	Treo 650 Phone/PDA	\$299.99	\$899.97	\$599.98		\$599.98	
	Sales Total	\$349.98	\$899.97	\$649.97	\$149.97	\$599.98	
Games	Nintendo DS				\$129.99		
	Nintendo Wii				\$1,319.98		
	PlayStation 2 Video Game	\$599.97	\$199.99	\$299.95	\$199.95	\$799.96	
	XBox 360 Video Game		\$299.99				
	Sales Total	\$599.97	\$499.98	\$299.95	\$1,649.92	\$799.96	

For example, you can use this code to produce the required custom formats for the sales totals, but not for the stoplight formatting:

```
public CellFormat getDataFormat(DataCellContext cxt)
{
    CellFormat cellFormat = new CellFormat(null, null, null);
    QDR qdr = cxt.getQDR();
    //Obtain a reference to the product category column.
    Object productCateg = qdr.getDimMember("ProductCategory");
    //Obtain a reference to the product column.
    Object product = qdr.getDimMember("ProductId");

    if (productCateg != null && productCateg.toString().equals("Sales Total"))
    {
        cellFormat.setStyle("background-color:#C0C0C0");
    }
    else if (product != null && product.toString().equals("Sales Total"))
    {
        cellFormat.setStyle("background-color:#C0C0C0");
    }
    return cellFormat;
}

public CellFormat getHeaderFormat(HeaderCellContext cxt)
{
    if (cxt.getValue() != null)
    {
        String header = cxt.getValue().toString();
        if (header.equals("Sales Total"))
    }
}
```

```

    {
        return new CellFormat(null, "background-color:#C0C0C0");
    }
}
return null;
}

```

The example includes the code for method expressions for both the `dataFormat` attribute and the `headerFormat` attribute of the `dvt:pivotTable` tag. If you want to include stoplight formatting in the pivot table, you might want to include the code from [Configuring Stoplight and Conditional Formatting Using CellFormat](#).

Configuring Stoplight and Conditional Formatting Using CellFormat

Stoplight and conditional formatting of the cells in a pivot table are examples of customizing the cell content. For this kind of customization, an application might prompt a user for a high value and a low value to be associated with the stoplight formatting.

Generally three colors are used as follows:

- Values equal to and above the high value are colored green to indicate they have no issues.
- Values above the low value but below the high value are colored yellow to warn that they are below the high standard.
- Values at or below the low value are colored red to indicate that they fall below the minimum acceptable level.

[Figure 27-36](#) shows data cells with stoplight formatting for minimum, acceptable, and below standards sales for States.

For example, you can use this code to perform stoplight formatting in a pivot table that does not display totals:

```

public CellFormat getDataFormat(DataCellContext cxt)
{
    //Use low and high values provided by the application.
    double low = m_rangeValues.getMinimum().doubleValue() * 100;
    double high = m_rangeValues.getMaximum().doubleValue() * 100;

    CellFormat cellFormat = new CellFormat(null, null, null);

    // Create stoplight format
    if (isStoptlightingEnabled())
    {
        String color = null;
        Object value = cxt.getValue();
        if (value != null && value instanceof Number)
        {
            double dVal = ((Number)value).doubleValue();
            if (dVal <= low)
            {
                color = "background-color:" + ColorUtils.colorToHTML(m_belowColor) + ";";
            }
            else if (dVal > low && dVal <= high)
            {
                color = "background-color:" + ColorUtils.colorToHTML(m_goodColor) + ";";
            }
            else if (dVal > high)

```

```
        {  
            color = "background-color:" + ColorUtils.colorToHTML(m_aboveColor) + ";";  
        }  
    }  
    cellFormat.setStyle(color);  
}  
return cellFormat;  
}
```

If you want to do stoplight formatting for a pivot table that displays totals, then you might want to combine the code from [Specifying a Cell Format](#) (which addresses rows with totals) with the code for stoplight and conditional formatting.

Using Gantt Chart Components

This chapter describes how to use the ADF Data Visualization `projectGantt`, `resourceUtilizationGantt`, and `schedulingGantt` components to display data in Gantt charts using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the technology stack, then you can also use data controls to create Gantt charts. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Gantt Chart Components](#)
- [Using the Gantt Chart Components](#)
- [Customizing Gantt Chart Tasks and Resources](#)
- [Customizing Gantt Chart Display Elements](#)
- [Adding Interactive Features to Gantt Charts](#)

About the Gantt Chart Components

An ADF DVT Gantt Chart is a visualization tool used to plan and track projects. It comprises two regions, containing a bar chart and a table, and shows resources or tasks in a defined and limited time frame. Gantt charts are useful for project management, both at an overview and a transactional level.

A Gantt chart shows resources or tasks in a time frame with a distinct beginning and end. Both regions of a Gantt chart present data differently, one displaying the Gantt chart data in a table, and the other displaying the Gantt chart data graphically with a resizable splitter between the two regions. The table and chart regions share the same data and selection model, supporting and synchronizing scrolling, and expanding and collapsing of rows between the two regions.

At runtime, Gantt charts provide interaction capabilities in the table region to the user such as entering data, expanding and collapsing rows, showing and hiding columns, navigating to a row, and sorting and totaling columns. In the chart region, users can drag a task to a new date, select multiple tasks to create dependencies, and extend or shorten the task date. A Gantt chart toolbar is available to support user operations such as changing or filtering the view of the data, and creating, deleting, cutting, copying, and pasting tasks.

Both Gantt chart regions are based on an ADF Faces tree table component. For information about ADF tree tables, including virtualization of rows, see [Using Tables, Trees, and Other Collection-Based Components](#).

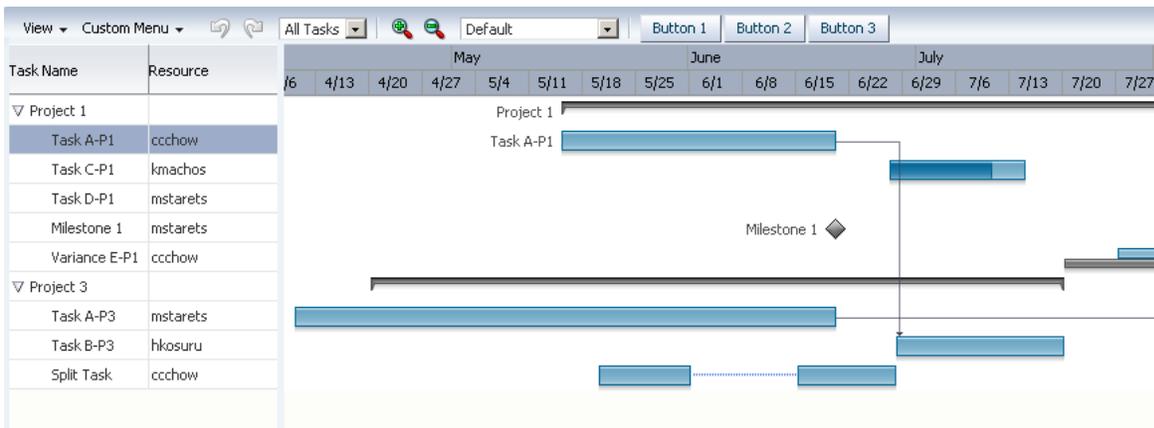
ADF Gantt chart components include a project Gantt chart (`projectGantt`), a resource utilization Gantt chart (`resourceUtilizationGantt`), and a scheduling Gantt chart (`schedulingGantt`).

Gantt Chart Component Use Cases and Examples

The Gantt chart provides the following types:

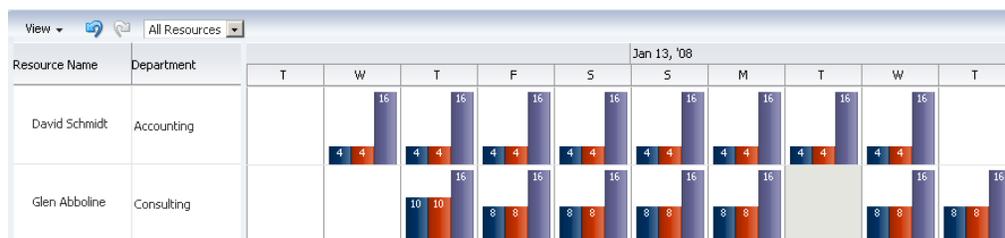
- Project Gantt chart: A project Gantt chart is used for project management. The chart lists tasks vertically and shows the duration of each task as a bar on a horizontal time line. It graphs each task on a separate line as shown in [Figure 28-1](#).

Figure 28-1 Project Gantt Chart for a Software Application



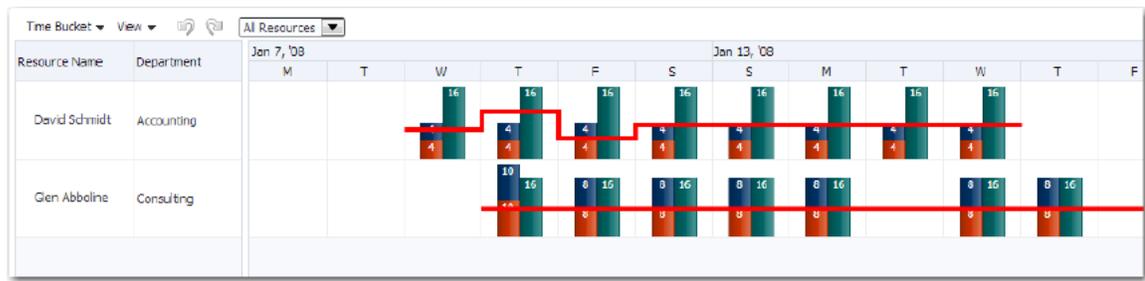
- Resource Utilization Gantt chart: A resource utilization Gantt chart graphically shows the metrics for a resource, for example, whether resources are over or under allocated. It shows resources vertically while showing their metrics, such as allocation and capacity on the horizontal time axis. [Figure 28-2](#) shows a resource utilization Gantt chart illustrating how many hours are allocated and utilized for a particular resource in a given time period.

Figure 28-2 Resource Utilization Gantt Chart for a Software Application



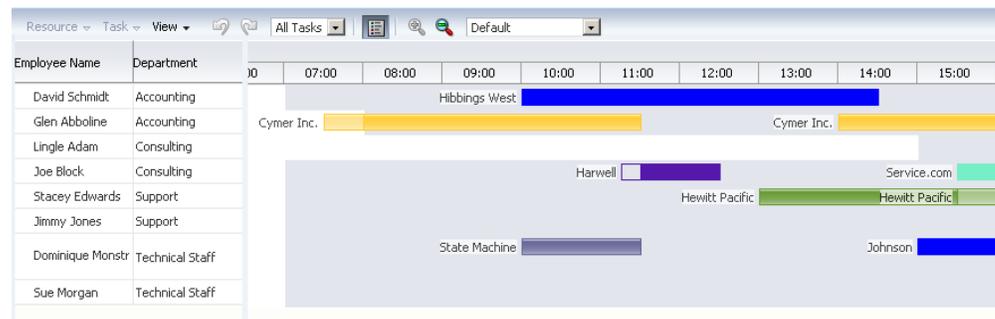
The metrics for a resource utilization Gantt chart can also be configured to display as stacked bars, or as a horizontal line. [Figure 28-3](#) shows a resource utilization Gantt chart illustrating a stacked bar representing hours utilized for two metrics, a vertical bar representing hours allocated, and a horizontal line representing a threshold metric that steps through the chart.

Figure 28-3 Resource Metrics Displayed as Stacked Bars and Stepped Line



- Scheduling Gantt chart: A scheduling Gantt chart is used for resource scheduling. The chart is based on manual scheduling boards and shows resources vertically, with corresponding activities on the horizontal time axis. Examples of resources include people, machines, or rooms. The scheduling Gantt chart uses a single line to graph all the tasks that are assigned to a resource as shown in [Figure 28-4](#).

Figure 28-4 Scheduling Gantt Chart for a Software Application



End User and Presentation Features

To understand how Gantt charts are used and can be customized, it is helpful to understand these elements and features.

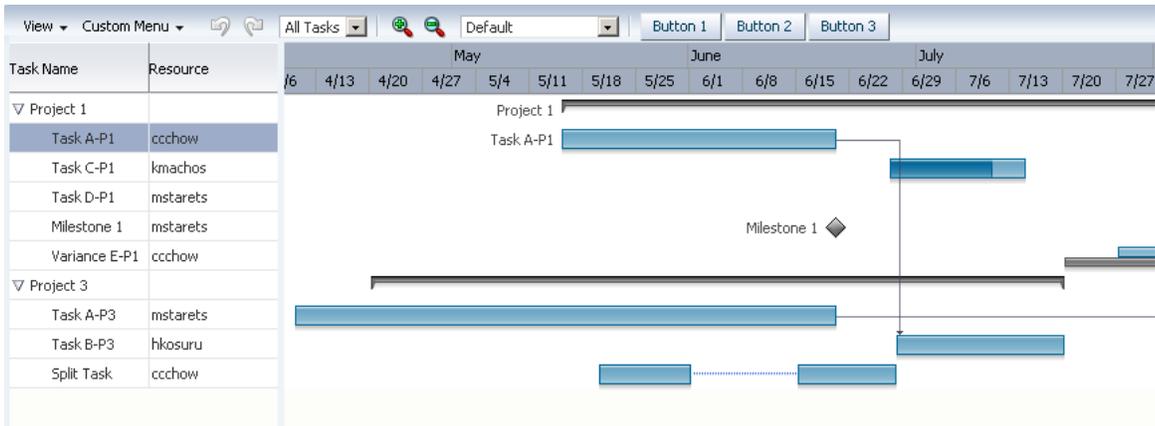
Gantt Chart Regions

A Gantt chart is composed of two regions:

- A table region that displays Gantt chart data attributes in a table with columns. The table region requires a minimum of one column, but you can define attributes for as many columns as desired. By default, all the columns that you define when you create a databound Gantt chart are visible in the table region although you can selectively cause one or more of these columns to be hidden.
- A chart region displays a bar graph of the Gantt chart data along a horizontal time axis. The time axis provides for major and minor settings to allow for zooming. The major setting is for larger time increments and the minor setting is for smaller time increments.

For example, in [Figure 28-5](#), the scheduling Gantt chart table region contains columns for Task Name and Resources, and the chart region graphs tasks on a time axis that shows weeks within months.

Figure 28-5 Project Gantt Chart Regions

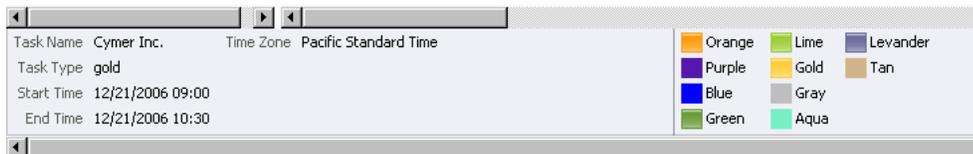


Information Panel

The optional information panel displays both the information region that displays text about a selected task, or metric about a selected resource, and the optional legend that displays task types in the area beneath the Gantt chart. You must configure a Gantt chart legend to enable the information panel.

Figure 28-6 shows an information panel for the scheduling Gantt chart in Figure 28-4 with information about a task selected in the chart region and the Gantt chart legend.

Figure 28-6 Scheduling Gantt Chart Information Panel



Toolbar

The Gantt chart toolbar allows users to perform operations on the Gantt chart. The toolbar is visible in the Gantt chart by default.

The toolbar is composed of two sections:

- **Menu bar:** The left section of the toolbar contains a set of menus for the Gantt chart. Each Gantt chart type has a set of default options. Figure 28-7 displays the menu bar, which is visible in the Gantt chart by default. It contains the Task, Edit, and View menus. You can change the visibility of the menu bar and customize menu items.

 **Note:**

The default View menu is available by right-clicking header cells in the Gantt table region, even when the menu bar items are hidden or disabled.

Figure 28-7 Sample Menu Bar for a Gantt Chart



By default, Gantt chart View menu items support one or more of the following operations:

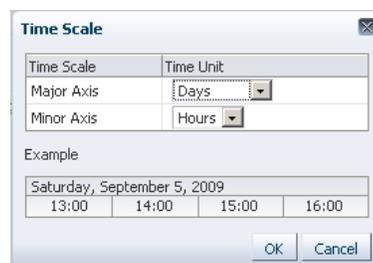
- Configuring the visibility of columns in the table region.
- Expanding and collapsing the display of hierarchical data in the table region.
- Hiding and showing dependency lines between tasks in the chart region.
- Hiding and showing the Gantt chart legend.
- Specify a specific date in the Gantt chart. [Figure 28-8](#) shows the View menu Go to Date dialog.

Figure 28-8 Go to Date Dialog



- Changing the time scale of the Gantt chart. [Figure 28-9](#) shows the View menu Time Scale dialog.

Figure 28-9 Time Scale Dialog



 **Note:**

The menu bar **View** menu items do *not* require that you write application code to make them functional. However, you must provide application code for any items that you want to use on the other menus.

- **Toolbar buttons:** The right section of the toolbar displays a set of action buttons for working with the Gantt chart. Each Gantt chart type has a set of default options. [Figure 28-10](#) shows a sample toolbar for a project Gantt chart.

Figure 28-10 Sample Toolbar for a Project Gantt Chart



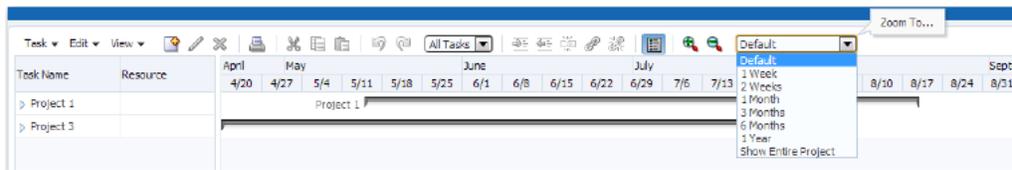
 **Note:**

In locales using right-to-left display, directional icons are displayed in reverse order.

You can use the toolbar to change the time display on the Gantt chart. You can zoom in and out on a time axis to display the chart region in different time units.

You can also use a specialized zoom-to-fit feature in which you select the amount of time that you want to display in the chart region without a need to scroll the chart. The Gantt chart time scale in the View menu automatically adjusts for the selected dates. [Figure 28-11](#) shows the zoom-to-fit toolbar option expanded for a project Gantt chart.

Figure 28-11 Zoom-to-Fit Toolbar Option



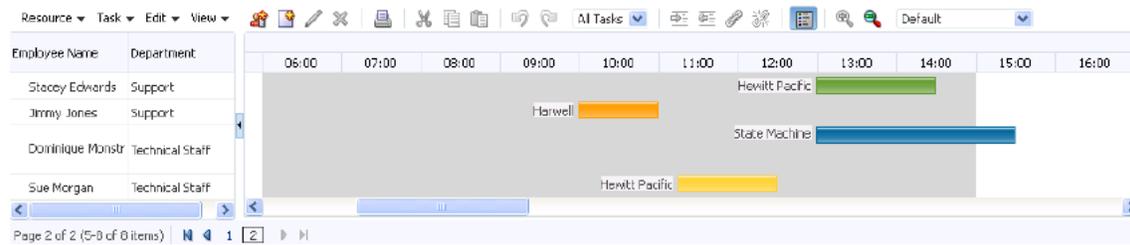
Scrolling, Zooming, and Panning

The Gantt chart design lets you perform horizontal scrolling of the table and the chart regions independently. This is especially helpful when you want to hold specific task or resource information constant in the table region while scrolling through multiple time periods of information in the chart region.

As an alternative to scrollbars, you can also display a horizontal page control that allows the user to select and navigate through a multiple page Gantt chart.

[Figure 28-12](#) shows a resource utilization Gantt chart configured with a horizontal page control.

Figure 28-12 Resource Utilization Gantt Chart Page Control



In addition to the toolbar zoom controls, users can also zoom in and out on the time scale of a Gantt chart by holding the Ctrl key and using the mouse scroll wheel. A tooltip displays to allow the user to keep track of the current level when zooming through multiple levels at a time. This is especially useful for users with a scroll wheel without a click function.

In project and scheduling Gantt charts, users can pan the chart area by dragging it vertically and horizontally using the cursor. A move cursor displays when the user clicks inside the chart area, other than on a task.

The Gantt chart also provides a user option to collapse and expand the display of the table region using an icon available in the vertical space between the two regions.

Showing Dependencies

When dependencies between tasks are specified, project and scheduling Gantt charts can optionally show dependency lines in the chart region. The option to display dependency lines is available as a **View** menu item. Additionally for project Gantt charts, you can show dependencies as a menu option for the predecessor or successor task using a dropdown menu at the beginning or end of the task with a dependency.

Figure 28-13 shows a project Gantt chart **View** menu with options for showing dependencies as lines, menu items, or not displayed.

Figure 28-13 Project Gantt Chart Show Dependencies Options

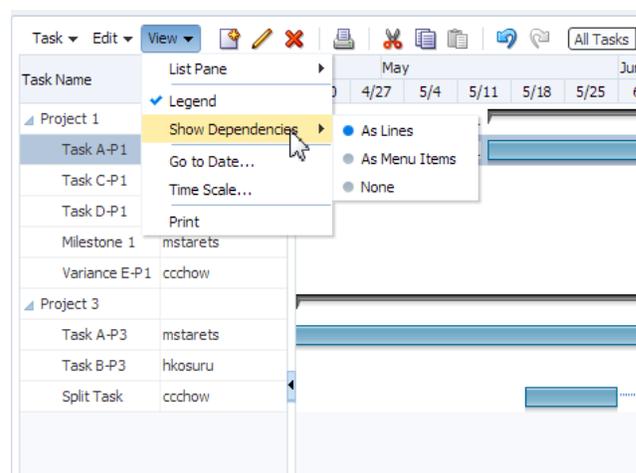
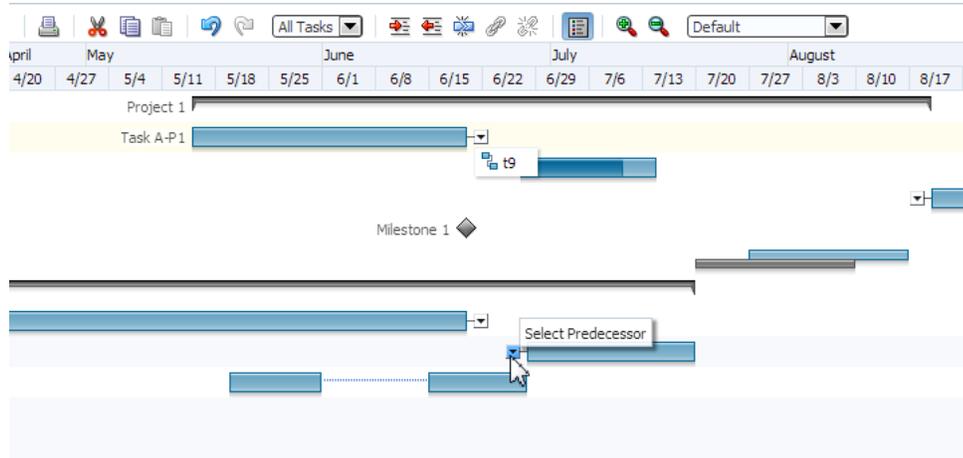


Figure 28-14 shows a project Gantt chart with dependency task menu items for a successor and predecessor task.

Figure 28-14 Project Gantt Chart Dependency Task Menu Items



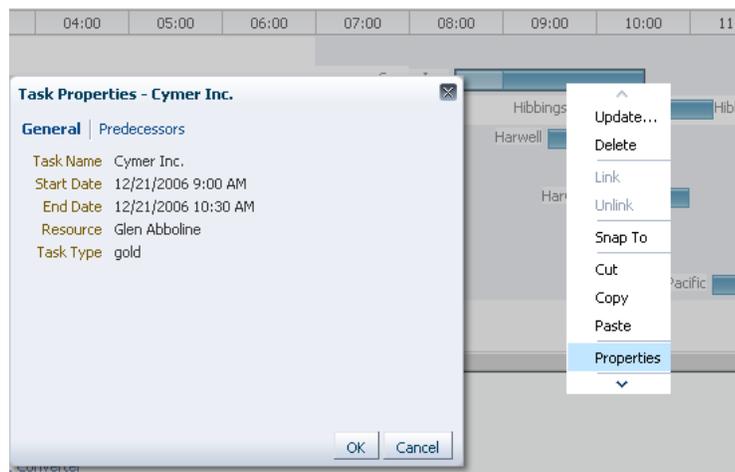
Context Menus

Right-clicking in the Gantt chart table or chart regions provides a context menu with a standard set of menu items. You can provide your own set of menu items by using the `tablePopupMenu` or `chartPopupMenu` facet.

For more information, see [Customizing Gantt Chart Context Menus](#).

Figure 28-15 shows a custom context menu item displayed for a scheduling Gantt chart task properties.

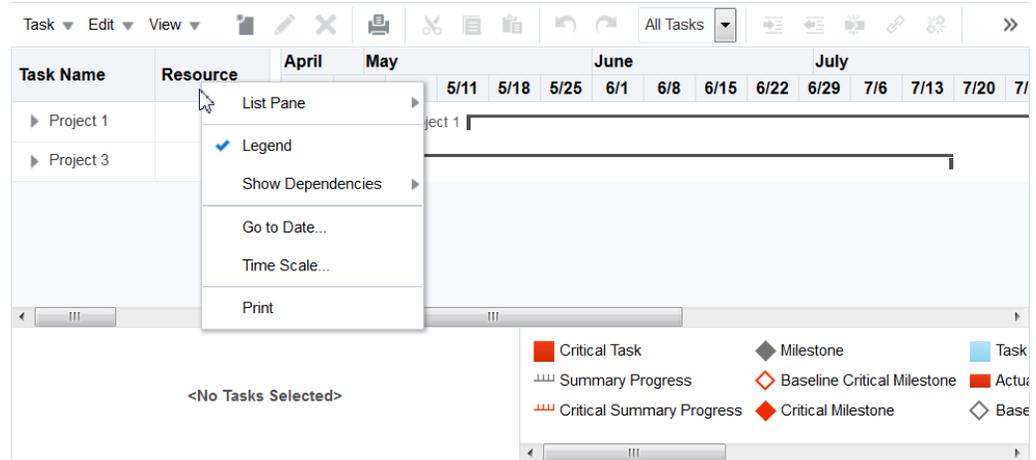
Figure 28-15 Custom Context Menu for Task



Right-clicking on the column headers in the table region of the Gantt chart displays the default View menu, even if the menu bar is hidden or disabled.

Figure 28-16 shows the default view menu in the context menu for column headers in the table region of a Project Gantt chart.

Figure 28-16 View Context Menu in Table Region



Row Selection

You can configure selection for no rows, for a single row, or for multiple rows in the chart region of a Gantt chart using the `rowSelection` attribute. You can select tasks for a project Gantt chart or resources for a scheduling or resource utilization Gantt chart. This setting allows you to execute logic against the selected tasks or resources. For example, you may want users to be able to select a resource and display a calendar based on that resource.

When the selected row in the table region changes, the component triggers a selection event. This event reports which rows were just selected or deselected. While the components handle selection declaratively, if you want to perform some logic on the selected rows, you need to implement code that can access those rows and then perform the logic. You can do this in a selection listener method on a managed bean. For more information, see [Performing an Action on Selected Tasks or Resources](#).

Note:

If you configure your component to allow multiple selection, users can select one row and then press the shift key to select another row, and all the rows in between will be selected. This selection will be retained even if the selection is across multiple data fetch blocks. Similarly, you can use the Ctrl key to select rows that are not next to each other.

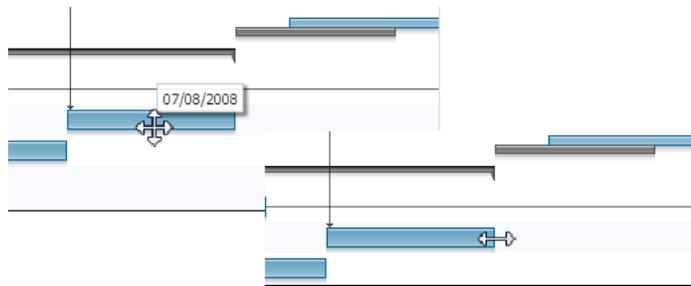
For example, if you configure your Gantt chart table region to fetch only 25 rows at a time, but the user selects 100 rows, the framework is able to keep track of the selection.

Editing Tasks

In project and scheduling Gantt charts, users can move or resize the task bar of an editable task in the chart region. Move the cursor over an editable task to display a move cursor with an informational popup about the location of the cursor. Click and drag the task bar with its associated beginning and end dates to reposition the task in the chart. If the beginning or ending date of a task is editable, a double-sided arrow is displayed at the start or end of the task bar. Click and drag the date to the desired location.

Figure 28-17 shows the move and resize cursors for an editable task in a project Gantt chart.

Figure 28-17 Move and Resize Cursors for an Editable Task Bar



Server-Side Events

When a user interaction involves a change in data, the Gantt chart processes the change by performing event handling and update of the data model. When configured for a Gantt chart, validation ensures that the data submitted meets basic requirements, for example, that a date is valid and does not fall into a nonworking time period. When validation fails, the update of the data model is omitted, and an error message is returned.

When a Gantt chart server-side event is fired, an event with validated information about the change is sent to the registered listener. The listener is then responsible for updating the underlying data model. A customized event handler can be registered by specifying a method binding expression on the `dataChangeListener` attribute of the Gantt chart component.

Server-side events supported by the Gantt chart include:

- Update of data in the table cells of the Gantt chart table region
- Create, update, delete, move, cut, copy, paste, indent, outdent of tasks
- Reassignment of resource by dragging the task bar from one row to another
- Drag the task bar to another date
- Extend the duration of a task
- Link or unlink tasks
- Select a row or multiple rows in the Gantt chart table region

- Undo or redo of user actions
- Double-click on a task bar

Users can filter the data in a Gantt chart using a dropdown list from the toolbar. You can create a custom filter.

Printing

The Gantt chart provides printing capability in conjunction with Apache or XML Publisher by generating PDF files. For more information, see [Printing a Gantt Chart](#).

Content Delivery

Gantt charts can be configured for how data is delivered from the data source. The data can be delivered to the Gantt chart task bars either immediately upon rendering, as soon as the data is available, or lazily fetch after the shell of the component has been rendered.

By default, Gantt charts support the delivery of content from the data source when it is available. The `contentDelivery` attribute is set to `whenAvailable` by default.

Gantt charts are virtualized, meaning not all data on the server is delivered to and displayed on the client. You can configure Gantt charts to fetch a certain number of rows or columns at a time from your data source based on date related values. Use `fetchSize` and `horizontalFetchSize` to configure fetch size.

Additional Functionality for Gantt Chart Components

You may find it helpful to understand other ADF Faces features before you implement your Gantt chart component. Additionally, once you have added a Gantt chart component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that Gantt chart components can use:

- **Partial page rendering:** You may want a Gantt chart to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- **Personalization:** Users can change the way the Gantt chart displays at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- **Accessibility:** Gantt chart components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- **Export to Excel:** You can export the table region of the project Gantt chart using `af:exportCollectionActionListener`. For more information, see [Exporting Data from Table, Tree, or Tree Table](#).
- **Content Delivery:** You configure your Gantt chart table region to fetch a certain number of rows at a time from your data source using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- **Automatic data binding:** If your application uses the technology stack, then you can create automatically bound Gantt charts based on how your ADF Business

Components are configured. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Gantt Chart Components

To use the ADF DVT Gantt chart component, add the Gantt chart component to a page using the Component Palette window. Then define the data collection for the Gantt chart and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

The data model for a Gantt chart can be either a tree (hierarchical) model or a collection model that contains a row set or flat list of objects. When you bind a Gantt chart to a data control, you specify how the collection in the data control maps to the node definitions of the Gantt chart. See *Creating Databound Gantt Chart and Timeline Components in Developing Fusion Web Applications with Oracle Application Development Framework*.

Data for a Project Gantt Chart

The data model for a project Gantt chart supports hierarchical data and uses `TreeModel` to access the data in the underlying list. The specific model class is `org.apache.myfaces.trinidad.model.TreeModel`.

The collection of objects returned by the `TreeModel` must have, at a minimum, the following properties:

- `taskId`: The ID of the task.
- `startTime`: The start time of the task.
- `endTime`: The end time of the task.

Optionally, the object could implement the `oracle.adf.view.faces.bi.model.Task` interface to ensure it provides the correct properties to the Gantt chart.

When binding the data to an ADF data control, the following node definitions are available in a project Gantt chart:

- Task node: Represents a collection of tasks. The task node definition has the following types of optional accessors:
 - subTask (available only for project Gantt chart)
 - splitTask
- Split task node: Represents a collection of split tasks. A split task node definition does not have accessors.
- Dependency node: Represents a collection of dependencies for a task. A dependency node definition does not have accessors.
- Recurring task node: Represents a collection of recurring tasks. A recurring task node definition does not have accessors.

Table 28-1 shows a complete list of data object keys for the project Gantt chart.

Table 28-1 Data Object Keys for Project Gantt

Data Object Key	Date Type and Description
actualEnd	Date. The actual end time for normal and milestone tasks.
actualStart	Date. The actual start time for normal and milestone tasks.
completedThrough	Date. Completed through for normal and summary tasks.
critical	Boolean. Specifies whether or not the task is critical for all tasks.
Dependency (node)	A list of dependencies for a task. Data object keys for dependencies include: <ul style="list-style-type: none"> • fromId: The ID of the task where the dependency begins. • toId: The ID of the task where the dependency ends. • type: The type of the dependency. Valid values are start-start, start-finish, finish-finish, finish-start, start-before, start-together, finish-after, and finish-together.
editsAllowed	Boolean. Specifies whether or not a task bar can be edited in the chart region.
endTime (required)	Date. The end time for all tasks.
icon1	String. The first icon associated with the task bar for all tasks. The icon might change depending on other attributes
icon2	String. The second icon associated with the tasks bar for all tasks.
icon3	String. The third icon associated with the tasks bar for all tasks.
iconPlacement	String. The alignment of the icon in the task bar for all tasks. Valid values are left (default), right, inside, start, end, innerLeft, innerRight, innerCenter, innerStart, innerEnd.
isContainer	Boolean. Specifies whether or not a node definition is a container.
label	String. The label associated with the task bar for all tasks.
labelPlacement	String. The alignment of the label in the task bar for all tasks. Valid values are left (default), right, inside, start, end, innerLeft, innerRight, innerCenter, innerStart, innerEnd.

Table 28-1 (Cont.) Data Object Keys for Project Gantt

Data Object Key	Date Type and Description
<code>percentComplete</code>	Integer. Percentage completed for normal and summary tasks.
Recurring tasks (node)	The list of recurring tasks for all tasks.
Split tasks (node)	The list of tasks without a continuous time line for all tasks.
<code>startTime</code> (required)	Date. The starting time for all tasks.
Subtasks (node)	An optional list of subtasks for all tasks.
<code>taskId</code> (required)	String. The unique identifier for all tasks.
<code>type</code>	String. The type of the tasks for all tasks.

Data for a Resource Utilization Gantt Chart

The data model for a resource utilization Gantt chart supports hierarchical data and uses `TreeModel` to access the data in the underlying list. The specific model class is `org.apache.myfaces.trinidad.model.TreeModel`.

The collection of objects returned by `TreeModel` must have, at a minimum, the following properties:

- `resourceId`: The ID of the task.
- `timeBuckets`: A collection of time bucket objects for this resource.

Optionally, the object could implement the `oracle.adf.view.faces.bi.model.Resource` interface to ensure it provides the correct properties to the Gantt chart.

The collection of objects returned by the `timeBuckets` property must also have the following properties:

- `time`: The date represented by the time bucket.
- `values`: A list of metrics for this resource.

When binding the data to an ADF data control, the following node definitions are available in a Resource Utilization Gantt chart:

- Resource node: Represents a collection of resources. The resource node definition has an optional `subResources` accessor that returns a collection of subresources for the current resource.
- Time bucket node: Represents a collection of time slots with metrics defined.
- Time bucket details node: Optional child accessor to the time bucket node that represents a collection of rows that would be rendered along with other metric values in the time bucket.

[Table 28-2](#) shows a complete list of data object keys for the resource utilization Gantt chart.

Table 28-2 Data Object Keys for Resource Utilization Gantt

Data Object Key	Data Type and Description
label	String. The label associated with the task bar.
labelAlign	String. The alignment of the label in the task bar. Valid values are <i>top</i> (default) and <i>inside</i> .
resourceId (required)	String. The unique identifier of a resource.
timeBuckets (required)	List. The list of tasks associated with a resource.
timeBucketDetail	List. The list of attributes associated with a resource.
time (required)	Date. The start time of the time bucket.
values (required)	Double. The values of the metrics.

Data for a Scheduling Gantt Chart

The data model for a scheduling Gantt chart supports hierarchical data and uses `TreeModel` to access the data in the underlying list. The specific model class is `org.apache.myfaces.trinidad.model.TreeModel`.

The collection of objects returned by `TreeModel` must have, at a minimum, the following properties:

- `resourceId`: The ID of the resource.
- `tasks`: A collection of task objects for this resource.

Optionally, the object could implement the `oracle.adf.view.faces.bi.model.ResourceTask` interface to ensure it provides the correct properties to the Gantt chart.

The collection of objects returned by the `tasks` property must also have the following properties:

- `taskId`: The ID of the task.
- `startTime`: The start time of the task.
- `endTime`: The end time of the task.

Optionally, a `backgroundBars` property can be set for each task object to customize the appearance of individual background bars for a given resource over a daily time interval, as in the case of a change in shift on a monthly, weekly, or even daily schedule. The collection of objects returned by the `backgroundBars` property must have the following properties:

- `startTime`: The start time of the task.
- `endTime`: The end time of the task.
- `type`: Linked to an instance of a `backgroundBarFormat` object that can be registered with the Gantt chart through the `TaskbarFormatManager`.

When binding the data to an ADF data control, the scheduling Gantt chart has a Resource node definition. The Resource node has the following types of accessors:

- `subResources`: Returns a collection of subresources for the current resource. This accessor is optional.
- `tasks`: Returns a collection of tasks for the current resource. This accessor is required. Tasks can also include a `splitTask` accessor.

Table 28-3 shows a complete list of data object keys for a scheduling Gantt chart.

Table 28-3 Data Object Keys for Scheduling Gantt Chart

Data Object Key	Data Type and Description
Dependency (node)	A list of dependencies for a task. Data object keys for dependencies include: <ul style="list-style-type: none"> • <code>fromId</code>: The ID of the task where the dependency begins. • <code>toId</code>: The ID of the task where the dependency ends. • <code>type</code>: The type of the dependency. Valid values are <code>start-start</code>, <code>start-finish</code>, <code>finish-finish</code>, <code>finish-start</code>, <code>start-before</code>, <code>start-together</code>, <code>finish-after</code>, and <code>finish-together</code>.
<code>endTime</code> (required)	Date. The end time for all the tasks.
<code>icon1</code>	String. The first icon associated with the task bar for all tasks. The icon might change depending on other attributes.
<code>icon2</code>	String. The second icon associated with the task bar for all tasks.
<code>icon3</code>	String. The third icon associated with the task bar for all tasks.
<code>iconPlacement</code>	String. The alignment of the icon in the task bar for all tasks. Valid values are <code>left</code> (default), <code>right</code> , <code>inside</code> , <code>inside_left</code> , <code>inside_right</code> , and <code>inside_center</code> . In locales using right-to-left display, <code>start</code> and <code>end</code> values are also supported.
<code>isContainer</code>	Boolean. Specifies whether or not a node definition is a container.
<code>label</code>	String. The label associated with the task bar for all tasks.
<code>labelPlacement</code>	String. The alignment of the label in the task bar for all tasks. Valid values are <code>left</code> (default), <code>right</code> , <code>inside</code> , <code>inside_left</code> , <code>inside_right</code> , and <code>inside_center</code> . In locales using right-to-left display, <code>start</code> and <code>end</code> values are also supported.
Recurring tasks (node)	A list of recurring tasks for all tasks.
<code>resourceId</code> (required)	String. The unique identifier of a resource.
Split tasks (node)	A collection of tasks without a continuous time line for all tasks.
<code>startTime</code> (required)	Date. The start time for all tasks.
<code>startupTime</code>	Date. The startup time before a task begins.
Tasks (node) (required)	A list of tasks associated with a resource.
<code>taskId</code> (required)	String. The unique identifier of the task for all tasks.
<code>taskType</code>	String. The type of the task for all tasks.
<code>workingDaysOfTheWeek</code>	Object. A list of the working days of the week.
<code>workingEndTime</code>	String. The work end time for the resource.
<code>workingStartTime</code>	String. The work start time for the resource.

Gantt Chart Tasks and Resources

Project and scheduling Gantt charts use predefined tasks with a set of formatting properties that describe how the tasks will be rendered in the chart area. All supported tasks must have a unique identifier.

The following describes the supported tasks and how they appear in a Gantt chart:

- **Normal:** The basic task type. It is a plain horizontal bar that shows the start time, end time, and duration of the task.
- **Summary:** The start and end date for a group of subtasks. A summary task cannot be moved or extended. Instead, it is the responsibility of the application to execute code to recalculate the start and end date for a summary task when the date of a subtask changes. Summary tasks are available only for the project Gantt chart.
- **Milestone:** A specific date in the Gantt chart. There is only one date associated with a milestone task. A milestone task cannot be extended but it can be moved. A milestone task is available only for the project Gantt chart.
- **Recurring:** A task that is repeated in a Gantt chart, each instance with its own start and end date. Individual recurring tasks can optionally contain a subtype. All other properties of the individual recurring tasks come from the task which they are part of. However, if an individual recurring task has a subtype, this subtype overrides the task type.
- **Split:** A task that is split into two horizontal bars, usually linked by a line. The time between the bars represents idle time due to traveling or down time.
- **Scheduled:** The basic task type for a scheduling Gantt chart. This task type shows the starting time, ending time, and duration of a task, as well as startup time if one is specified.

For normal, summary, and milestone tasks, additional attributes are supported that would change the appearance and activity of a task. These style attributes include:

- `percentComplete`, `completedThrough`: An extra bar would be drawn to indicate how far the task is completed. This is applicable to normal and summary task types.
- `critical`: The color of the bar would be changed to red to mark it as critical. This is applicable to normal, summary, and milestone task types.
- `actualStart` and `actualEnd`: When these attributes are specified, instead of drawing one bar, two bars are drawn. One bar indicates the base start and end date, the other bar indicates the actual start and end date. This is applicable to normal and milestone task types.

Figure 28-18 displays a legend that shows common task types in a project Gantt chart.

Figure 28-18 Project Gantt Chart Legend for Task Types

Baseline	Summary Progress	Summary	Baseline Critical Milestone	Task	Actual Summary Progress
Task Progress	Critical Task	Critical Progress	Milestone	Critical Summary	Actual Progress
Actual Critical Task	Baseline Milestone	Critical Milestone	Actual Critical Progress	Actual	

Resource utilization Gantt charts graphically show resources vertically while displaying their metrics, such as allocation and capacity on the horizontal time axis. The metrics

displayed for the tasks in a time bucket are rendered as specified by the `TaskbarFormat` class and can be one of three types:

- `BAR` (default): Render the task as a vertical bar.
- `STACKED`: Render the task as a bar stacked on the previous bar.
- `STEPPED_LINE`: Render the task as a horizontal line.

Configuring Gantt Charts

The three Gantt chart components beginning with the prefix `dvt:` for each Gantt chart tag name indicates that the tag belongs to the ADF Data Visualization Tools (DVT) tag library:

- `projectGantt`
- `resourceUtilizationGantt`
- `schedulingGantt`

All Gantt chart components support the child tag `ganttLegend` to provide an optional legend in the information panel of a Gantt chart. Some menu bar and toolbar functions may or may not be available depending on whether the Gantt legend is specified.

In the Gantt chart table region, the ADF Faces `af:column` tag is used to specify the header text, icons and alignment for the data, the width of the column, and the data bound to the column. To display data in hierarchical form, a `nodeStamp` facet specifies the primary identifier of an element in the hierarchy. For example, the "Task Name" column might be used as the `nodeStamp` facet for a project Gantt chart. The example below shows sample code for a project Gantt chart with "Task Name" as the `nodeStamp` facet, with columns for Resource, Start Date, and End Date.

```
<dvt:projectGantt id="projectChart1" startTime="2008-04-12"
    endTime="2009-04-12"
    value="#{project.model}"
    var="task">
  <f:facet name="major">
    <dvt:timeAxis scale="months"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="weeks"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column headerText="Task Name">
      <af:outputText value="#{task.taskName}"/>
    </af:column>
  </f:facet>
  <af:column headerText="Resource">
    <af:outputText value="#{task.resourceName}"/>
  </af:column>
  <af:column headerText="Start Date">
    <af:outputText value="#{task.startTime}"/>
  </af:column>
  <af:column headerText="End Date">
    <af:outputText value="#{task.endTime}"/>
  </af:column>
</dvt:projectGantt>
```

In addition to the `nodeStamp` facet, other facets are used for customizations by the Gantt chart components. [Table 28-4](#) shows the facets supported by Gantt chart components.

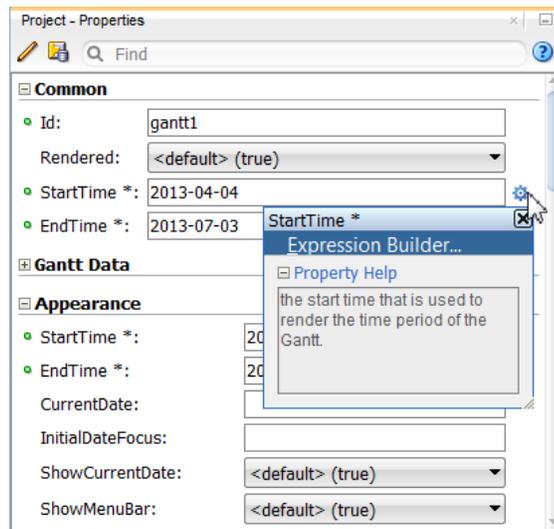
Table 28-4 Facets Supported by Gantt Chart Components

Facet	Description
<code>chartPopupMenu</code>	Specifies the component to use to identify additional controls to appear in the context menu of the chart region. Must be an <code>af:popup</code> component.
<code>customPanel</code>	Specifies the component to use to identify controls to appear in the custom tab of the task properties dialog.
<code>major</code>	Specifies the component to use to identify the major time axis. Must be a <code>dvt:timeAxis</code> component.
<code>menuBar</code>	Specifies the component to use to identify additional controls to appear in the Gantt menu bar. Must be an <code>af:menu</code> component.
<code>minor</code>	Specifies the component to use to identify the minor time axis. Must be a <code>dvt:timeAxis</code> component.
<code>nodeStamp</code>	Specifies the component to use to stamp each element in the Gantt chart. Only certain types of components are supported, including all components with no activity and most components that implement the <code>EditableValueHolder</code> or <code>ActionSource</code> interfaces. Must be an <code>af:column</code> component.
<code>tablePopupMenu</code>	Specifies the component to use to identify additional controls to appear in the context menu of the table region. Must be an <code>af:popup</code> component.
<code>toolbar</code>	Specifies the component to use to identify additional controls to appear in the Gantt toolbar. Must be an <code>af:toolbar</code> component.

How to Add a Gantt Chart to a Page

When you are designing your page using simple UI-first development, you use the Components window to drag and drop a project, resource utilization, or scheduling Gantt chart component onto a JSF page. Once the Gantt chart is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the Gantt chart.

In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. [Figure 28-19](#) shows the dropdown menu for a project Gantt chart component `startTime` attribute.

Figure 28-19 Project Gantt Chart startTime Attribute Dropdown Menu**Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a Gantt chart and the binding will be done for you. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

To add a Gantt chart to a page:

1. In the ADF Data Visualizations page of the Components window, from the Gantt chart panel, drag and drop a **Project**, **Resource Utilization**, or **Scheduling** Gantt chart onto the page to open the Create Gantt chart dialog.

Optionally, use the dialog to bind the Gantt chart by selecting **Bind Data Now** and entering or navigating to the ADF data control or ADF managed bean that represents the data you wish to display on the Gantt chart. If you choose this option, the data binding fields in the dialog will be available for editing. Otherwise, click **OK** to add the component to the page. For help with the dialog, press F1 or click **Help**.

2. In the Properties window, view the attributes for the Gantt chart. Use the help button to display the complete tag documentation for the `projectGantt`, `resourceUtilizationGantt`, or `schedulingGantt` component.

3. Expand the **Common** section. Use this section to set the following attributes:
 - **StartTime**: Enter the start time used to render the time period of the Gantt chart.
 - **EndTime**: Enter the end time used to render the time period of the Gantt chart.
4. Expand the **Gantt Data** section. Use this section to set the following attributes:
 - **Value**: Specify the data model, which must be of type `org.apache.myfaces.trinidad.model.TreeModel`, using an EL Expression.
 - **Var**: Specify the variable used to reference each element of the Gantt chart data collection. Once this component has completed rendering, this variable is removed, or reverted back to its previous value.
5. Expand the **Appearance** section. Use this section to set the following attributes:
 - **RowBandingInterval**: Specify how many consecutive rows form a row group for the purposes of color banding. By default, this is set to 0, which displays all rows with the same background color. Set this to 1 if you want to alternate colors.
 - **ShowMenuBar**: Specify whether or not the menu bar should be shown in the Gantt chart. If this attribute is set to **false**, then any custom menu bar items specified in the `menuBar` facet will also be hidden.
 - **ShowToolBar**: Specify whether or not the toolbar should be shown in the Gantt chart. If this attribute is set to **false**, then any custom toolbar buttons specified in the `toolbar` facet will also be hidden.
 - **Summary**: Enter a description of the Gantt chart. This description is accessed by screen reader users
6. Expand the **Behavior** section. Use this section to set the following attributes:
 - **InitiallyExpandAll**: Specifies whether or not all the rows should be initially expanded.
 - **FetchSize**: Use to specify the number of rows in a data fetch block. The default value for rows is 25. For more information about content delivery to Gantt charts, see [Content Delivery](#).
 - **FeaturesOff**: Enter a space delimited list of end user features to disable at runtime. The valid values will depend upon the type of Gantt chart.
7. Expand the **Other** section. Use this section to set the following attributes:
 - **TableColumnStretching**: Use to indicate the type of stretching to apply to the columns in the table region of the Gantt chart. Valid values include the following:
 - `none` (default): Use for optimal performance of the Gantt chart.
 - `last`: Use to stretch the last column to fill up any unused space inside of the viewport.
 - `blank`: Use to automatically insert an empty blank column stretched to span the entire wide of the table region.
 - `column`: Use to stretch a specific leaf (non-group) column to fill up any unused space inside of the column. Append the Id of the column to be stretched to this value, for example:

```
column:ColId
```

- **multiple:** Use to stretch more than one column. You can set the widths in the columns as percentages. For more information, see the tag documentation for `af:column`.

 **Note:**

Row headers and frozen columns will not be stretched in order to prevent inaccessibility to the scrollable data body of the table region.

- **TableActiveRowKey:** Use to determine the currently active row on the table region. By default, the value of this attribute is the first visible row of the table region. When the table region is refreshed, that component scrolls to bring the active row into view, if it is not already visible. When the user clicks on a row to edit its contents, that row becomes the active row.

What Happens When You Add a Gantt Chart to a Page

When you use the Components window to create a Gantt chart, JDeveloper inserts code in the JSF page.

The example below shows the code inserted in the JSF page for a project Gantt chart.

```
<dvt:projectGantt startTime="2011-03-20" endTime="2011-06-19" var="row"
id="pg1">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks" id="ta5"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days" id="ta6"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column sortable="false" headerText="col1" id="c11">
      <af:outputText value="#{row.col1}" id="ot11"/>
    </af:column>
  </f:facet>
  <af:column sortable="false" headerText="col2" id="c12">
    <af:outputText value="#{row.col2}" id="ot12"/>
  </af:column>
  <af:column sortable="false" headerText="col3" id="c13">
    <af:outputText value="#{row.col3}" id="ot13"/>
  </af:column>
  <af:column sortable="false" headerText="col4" id="c14">
    <af:outputText value="#{row.col4}" id="ot14"/>
  </af:column>
  <af:column sortable="false" headerText="col5" id="c15">
    <af:outputText value="#{row.col5}" id="ot15"/>
  </af:column>
</dvt:projectGantt>
```

Customizing Gantt Chart Tasks and Resources

Once you have added an ADF DVT Gantt Chart to your JSF page, you can create a new task type, configure tasks to display as a stacked bar or horizontal line, and display the attribute details for a task.

Creating a New Task Type

A task type is represented visually as a bar in the chart region of a Gantt chart. Task types can be created in one of three ways.

You can create a new task type by:

- Defining the task type style properties in the JSF page or in a separate CSS file.
- Defining a `TaskbarFormat` object and registering the object with the `taskbarFormatManager`.
- Modifying the properties of a predefined task type by retrieving the associated `TaskbarFormat` object and updating its properties through a `set` method.

The `TaskBarFormat` object exposes the following properties:

- Fill color
- Fill image pattern
- Border color
- Images used for a milestone task
- Images used for the beginning and end of a summary task

For tasks that have more than one bar, such as a split or recurring task, properties are defined for each individual bar.

The example below shows sample code to define the properties for a custom task type in the JSF page.

```
<af:document>
  <f:facet name="metaContainer">
    <f:verbatim>
      <![CDATA[
        <style type="text/css">
          .onhold
        {
          background-image:url('images/Bar_Image.png');
          background-repeat:repeat-x;
          height:13px;
          border:solid 1px #000000;
        }
      </style>
    </f:verbatim>
  </f:facet>
```

The example below shows sample code to define a `TaskbarFormat` object fill and border color and register the object with the `taskbarFormatManager`.

```
TaskbarFormat _custom = new TaskbarFormat("Task on hold", null, "onhold",
null);
//
_gantt.getTaskbarFormatManager().registerTaskbarFormat("FormatId",
_custom);
TaskbarFormat _custom = new TaskbarFormat("Task on hold", "#FF00FF", null,
"#00FFDD", 13);
// _gantt.getTaskbarFormatManager().registerTaskbarFormat("FormatId",
_custom);
```

Configuring Stacked Bars in Resource Utilization Gantt Charts

In a resource utilization Gantt chart, a time bucket displays the unit allocated and used for a resource for a specified time period. By default, these units are rendered as vertical bars in the chart region. You can also configure the graphical display to stack two or more of the bars together.

For example, the resource utilization Gantt chart in [Figure 28-3](#) stacks the `RUN` and `SETUP` resource metrics into a vertical bar next to the `AVAILABLE` resource metric bar.

To configure stacked bars in a resource utilization Gantt chart, use the `setDisplayAs()` method to update the `TaskbarFormat` object. Specifying a `STACK` display renders the task as a metric stacked on the previous metric. The example below shows a managed bean that configures the `RUN` metric to stack on the previous metric, `SETUP`.

```
public class ResourceUtilizationGantt
{
    private TreeModel m_model;
    public String[] getMetrics()
    {
        return new String[]{"SETUP", "RUN", "AVAILABLE"};
    }

    public TaskbarFormatManager getTaskbarFormatManager()
    {
        TaskbarFormatManager _manager = new TaskbarFormatManager();
        TaskbarFormat _format = TaskbarFormat.getInstance("Run Hours",
            UIResourceUtilizationGantt.MIDNIGHT_BLUE_FORMAT);
        _format.setStacked(true);

        _manager.registerTaskbarFormat("SETUP",
TaskbarFormat.getInstance("Setup
        Hours", UIResourceUtilizationGantt.BRICK_RED_FORMAT));
        _manager.registerTaskbarFormat("RUN", _format);
        _manager.registerTaskbarFormat("AVAILABLE",
            TaskbarFormat.getInstance("Available Hours",
                UIResourceUtilizationGantt.TEAL_FORMAT));
        return _manager;
    }
}
```

Configuring a Resource Capacity Line

In addition to displaying the resource metrics as vertical bars in the chart region of a resource utilization Gantt chart, you can configure a metric to display as a horizontal line in the chart region. This is useful for displaying capacity metrics, such as a resource threshold level.

For example, [Figure 28-3](#) shows a resource utilization Gantt chart with a capacity line displaying a threshold metric across the stacked RUN and SETUP metrics, and the ALLOCATED metric bars.

To configure a resource capacity line in a resource utilization Gantt chart, use the `setDisplayAs()` method to update the `TaskbarFormat` object. Specifying a `STEPPED_LINE` display renders the task as a horizontal line over the vertical bars, stepping through each metric. The example below shows the managed bean that configures the resource metric `THRESHOLD` to step through the vertical bar metrics.

```
public class ResourceUtilizationGanttSteppedLine
{
    private TreeModel m_model;
    public String[] getMetrics()
    {
        return new String[]{"SETUP", "RUN", "AVAILABLE", "THRESHOLD"};
    }
    public TaskbarFormatManager getTaskbarFormatManager()
    {
        TaskbarFormatManager _manager = new TaskbarFormatManager();
        TaskbarFormat _format = TaskbarFormat.getInstance("Run Hours",
            UIResourceUtilizationGantt.MIDNIGHT_BLUE_FORMAT);
        _format.setStacked(true);

        _manager.registerTaskbarFormat("SETUP",
            TaskbarFormat.getInstance("Setup
            Hours", UIResourceUtilizationGantt.BRICK_RED_FORMAT));
        _manager.registerTaskbarFormat("RUN", _format);
        _manager.registerTaskbarFormat("AVAILABLE",
            TaskbarFormat.getInstance("Available Hours",
            UIResourceUtilizationGantt.TEAL_FORMAT));

        MetricFormat _threshold = new MetricFormat("threshold", "#FF0000",
            null,
            "#FF0000", MetricFormat.Display.STEPPED_LINE);
        _manager.registerTaskbarFormat("THRESHOLD", _threshold);
        return _manager;
    }
}
```

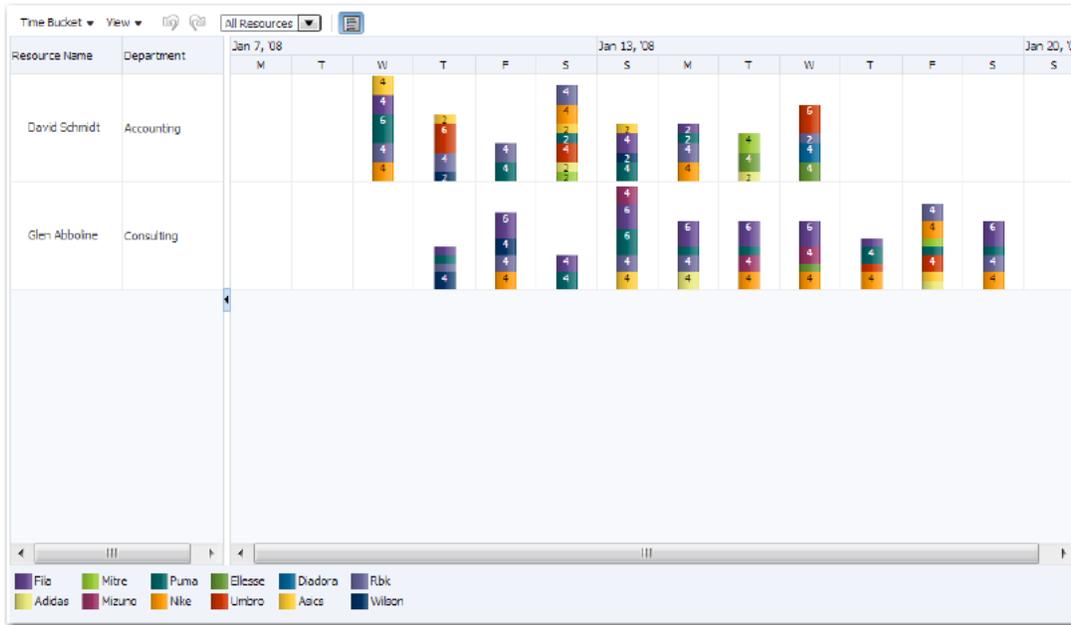
Displaying Resource Attribute Details

By default, the time buckets in resource utilization Gantt charts display a fixed size metric.

For example, the Gantt chart in [Figure 28-20](#) displays stacked RUN and SETUP metrics, and ALLOCATED metric bars for a table of company resources.

You may wish to break out the detail for a time bucket metric by attributes associated with the resource. For example, the resource utilization Gantt chart in [Figure 28-20](#) illustrates the detail values for a BUDGET metric by actual values for time spent on each product.

Figure 28-20 Resource Utilization Gantt Chart Metric Attribute Details



To configure the display of the attribute details for a resource, you will need add a child `timeBucketDetails` accessor to the page definition file. The example below shows the sample code for adding the accessor to the page definition file.

```
<nodeDefinition DefName="model.GanttRugResourceAppView" type="Resources">
  <AttrNames>
    <Item Value="ResourceId" type="resourceId"/>
  </AttrNames>
  <Accessors>
    <Item Value="GanttRugTimebucketAppView" type="timeBuckets"/>
  </Accessors>
</nodeDefinition>
<nodeDefinition type="TimeBuckets"
DefName="model.GanttRugTimebucketAppView">
  <AttrNames>
    <Item type="time" Value="StartDate"/>
    <Item type="metric" Value="Budget"/>
  </AttrNames>
  <Accessors>
    <Item Value="GanttRugProductAppView" type="timeBucketDetails"/>
  </Accessors>
</nodeDefinition>
<nodeDefinition type="TimeBucketDetails">
  <AttrNames>
    <Item type="metric" Value="Actual"/>
    <Item type="format" Value="Product"/>
  </AttrNames>
</nodeDefinition>
```

```
</AttrNames>
</nodeDefinition>
```

To configure the attribute details for a resource in a resource utilization Gantt chart, use the `setDisplayAs()` method to update the `TaskbarFormat` object. The example below shows the managed bean that configures the attribute detail metrics for the BUDGET resource bar.

```
public class ResourceUtilizationGanttAttributeDetail
{
    private TreeModel m_model;

    public String[] getMetrics()
    {
        return new String[] {};
    }

    public TaskbarFormatManager getTaskbarFormatManager()
    {
        TaskbarFormatManager _manager = new TaskbarFormatManager();

        _manager.registerTaskbarFormat("Wilson",
            TaskbarFormat.getInstance("Wilson",
                UIResourceUtilizationGantt.MIDNIGHT_BLUE_FORMAT));
        _manager.registerTaskbarFormat("Umbro",
            TaskbarFormat.getInstance("Umbro",
                UIResourceUtilizationGantt.BRICK_RED_FORMAT));
        _manager.registerTaskbarFormat("Rbk",
            TaskbarFormat.getInstance("Rbk",
                UIResourceUtilizationGantt.LAVENDER_FORMAT));
        _manager.registerTaskbarFormat("Puma",
            TaskbarFormat.getInstance("Puma",
                UIResourceUtilizationGantt.TEAL_FORMAT));
        ...
        return _manager;
    }

    public TreeModel getModel()
    {
        if (m_model == null)
            m_model =
                SampleModelFactory.getResourceUtilizationGanttAttributeDetailModel();

        return m_model;
    }
}
```

Configuring Background Bars in Scheduling Gantt Charts

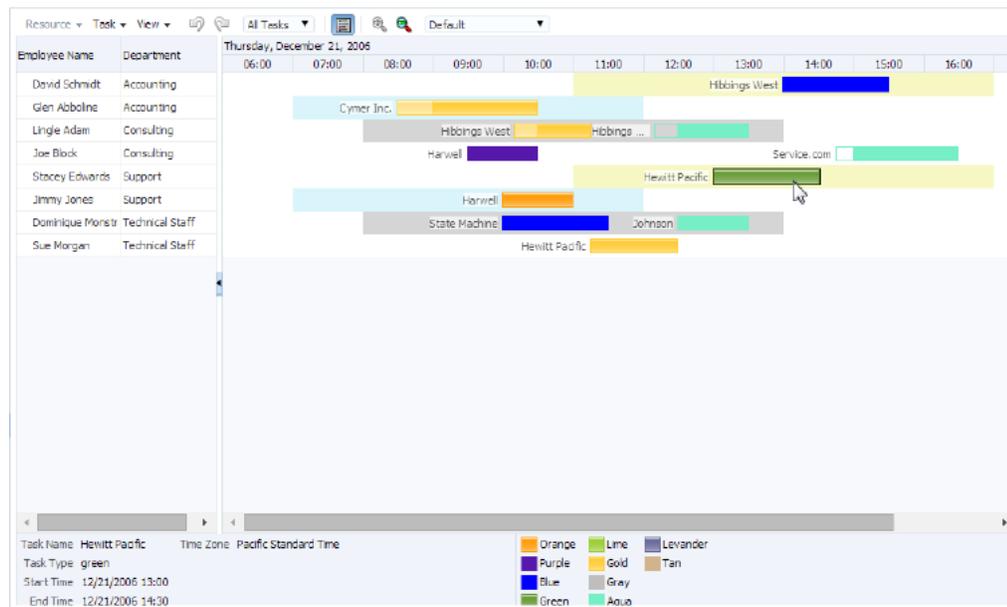
Scheduling Gantt charts use a single line to graph all the tasks assigned to a resource. You can configure a working time attribute that applies a specified background color to a given resource over a daily time interval. This time interval is repeated across all

working days of the week, and can be different for any given resource. However, if a resource's working time interval changes on a monthly, weekly, or even daily schedule, you can configure a background bar to represent that individual schedule.

To configure a background bar to a scheduling Gantt chart, set a `backgroundBars` property on the Gantt chart's model, and for each object contained within the collection, specify a `startTime`, `endTime`, and `type`. Each type can be linked to an instance of a `backgroundBarFormat` object registered with the `TaskbarFormatManager`. For more information, see [Creating a New Task Type](#).

The scheduling Gantt chart in [Figure 28-21](#) shows an individualized schedule for each resource represented as a background bar.

Figure 28-21 Scheduling Gantt Chart with Background Bars



To configure the background bars details for a resource in a scheduling Gantt chart, use the `setDisplayAs()` method to update the `TaskbarFormat` object. The example below shows the managed bean that configures the background bar details for each resource.

```
public class SchedulingGanttBackgroundBars extends SchedulingGanttBase
{
    private TreeModel m_model;
    public TreeModel getModel()
    {
        if (m_model == null)
            m_model =
SampleModelFactory.getSchedulingGanttBackgroundBarsModel();
        return m_model;
    }
    public String[] getLegendKeys()
    {return new String[]{"taskName", "taskType", "startTime",
        "endTime", "%timezone%"};
    }
}
```

```

public String[] getLegendLabels()
    {return new String[]{"Task Name", "Task Type",
        "Start Time", "End Time", "Time Zone"};
    }

public TaskbarFormatManager getTaskbarFormatManager()
{
    TaskbarFormatManager _manager = new TaskbarFormatManager();
    //create and register new colors
    _manager.registerBackgroundBarFormat("fillColor",
        new BackgroundBarFormat("fillColor", "#f6f7c3", null));
    _manager.registerBackgroundBarFormat("fillColor2",
        new BackgroundBarFormat("fillColor2", "#d9f4fa", null));
    //register predefined colors
    _manager.registerTaskbarFormat("gold",
TaskbarFormat.getInstance("Gold",
        UISchedulingGantt.GOLD_FORMAT));
    _manager.registerTaskbarFormat("green",
TaskbarFormat.getInstance("Green",
        UISchedulingGantt.GREEN_FORMAT));
    _manager.registerTaskbarFormat("orange",
        TaskbarFormat.getInstance("Orange",
        UISchedulingGantt.ORANGE_FORMAT));
    _manager.registerTaskbarFormat("levander",
        TaskbarFormat.getInstance("Levander",
        UISchedulingGantt.LAVENDER_FORMAT));
    _manager.registerTaskbarFormat("lime",
TaskbarFormat.getInstance("Lime",
        UISchedulingGantt.LIME_FORMAT));
    //create and register new colors
    _manager.registerTaskbarFormat("blue",
        new TaskbarFormat("Blue", "#0000FF", null, "#0000FF", 13));
    _manager.registerTaskbarFormat("purple",
        new TaskbarFormat("Purple", "#5518AB", null, "#5518AB", 13));
    _manager.registerTaskbarFormat("aqua",
        new TaskbarFormat("Aqua", "#76EEC6", null, "#76EEC6", 13));
    _manager.registerTaskbarFormat("gray",
        new TaskbarFormat("Gray", "#BEBEBE", null, "#BEBEBE", 13));
    _manager.registerTaskbarFormat("tan",
        new TaskbarFormat("Tan", "#D2B48C", null, "#D2B48C", 13));
    return _manager;
}
}

```

The example below shows sample code on the JSF page for the scheduling Gantt chart shown in [Figure 28-21](#).

```

<dvt:schedulingGantt id="schedulingGanttBackgroundBars"
    startTime="2006-12-21 06:00"
    endTime="2006-12-21 18:00"
    value="#{schedulingGanttBackgroundBars.model}"
    var="resourceObj"
    taskbarFormatManager="#{schedulingGanttBackgroundBars.
        taskbarFormatManager}"

```

```

summary="Scheduling Gantt Background Bars Demo">
<f:facet name="major">
  <dvt:timeAxis scale="days" id="ta1"/>
</f:facet>
<f:facet name="minor">
  <dvt:timeAxis scale="hours" id="ta2"/>
</f:facet>
<f:facet name="nodeStamp">
  <af:column headerText="Employee Name" id="c1">
    <af:outputText value="#{resourceObj.resourceName}" id="ot1"/>
  </af:column>
</f:facet>
  <af:column headerText="Department" id="c2">
    <af:outputText value="#{resourceObj.department}" id="ot2"/>
  </af:column>
  <dvt:ganttLegend keys="#{schedulingGanttBackgroundBars.legendKeys}"
labels="#{schedulingGanttBackgroundBars.legendLabels}"
  id="gll"/>
</dvt:schedulingGantt>

```

Customizing Gantt Chart Display Elements

ADF DVT Gantt charts support a number of configurations. You can customize a Gantt chart to display nonworking days of the week, turn off user interaction features, specify the time axes, add and customize a legend, customize toolbars and context menus, and configure a custom data filter.

Customizing Gantt Chart Toolbars and Menus

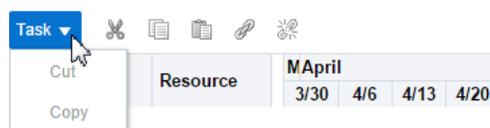
The Gantt chart toolbar subcomponent allows users to perform operations on the Gantt chart. The left section of the toolbar is a menu bar that contains a set of default menu options for each Gantt chart type. The right section of the toolbar displays a set of default action buttons for working with each Gantt chart type.

You can customize Gantt chart menus and toolbars by supplying your own items, and adding, removing, and rearranging items within the toolbar or menu. Use the `toolbarLayout` and `menuBarLayout` attributes of the Gantt chart component to specify the contents and arrangement of buttons and menu items as a set of space-separated Strings. Submenus are designated by using brackets in the `menuBarLayout` attribute as in:

```
taskMenu [cut copy]
```

For example, [Figure 28-22](#) shows a customized toolbar and menu bar for a project Gantt chart.

Figure 28-22 Project Gantt Chart Custom Toolbar and Menu Bar



The example below shows sample code for specifying the custom toolbar and menu bar in [Figure 28-22](#):

```
<dvt:projectGantt id="gantt1"
value="#{bindings.OrderEmployee1.projectGanttModel}"
    dataChangeListener="#{bindings.OrderEmployee1.
        projectGanttModel.processDataChanged}"
    var="row" startTime="2012-07-01" endTime="2013-05-15"
    initiallyExpandAll="true"
    summary="Project Gantt chart for shipping orders"
    toolBarLayout="cut copy paste link unlink"
    menuBarLayout="editTask [cut copy]">
    ...
</dvt:projectGantt>
```

[Table 28-5](#) shows the valid values for the `toolBarLayout` attribute.

Table 28-5 Valid Values for ToolBarLayout Attributes

Value	Feature Disabled
all	Show all standard toolbar items for all Gantt charts.
clipboard	Cut, Copy, and Paste tasks for all Gantt charts.
createTask	Create Task toolbar task for project and scheduling Gantt charts.
copy	Copy task for all Gantt charts.
cut	Cut task for all Gantt charts.
delete	Delete task for all Gantt charts.
edit	Changes to the data model for all Gantt charts.
filter	Data filter operation for all Gantt charts.
indent	Indent task for project and scheduling Gantt charts.
indentGroup	Indent group of tasks for project and scheduling Gantt charts.
indentSplitLinkGroup	Indent ////.
legend	Hide and Show legend toolbar items for all Gantt charts.
link	Link task for project and scheduling Gantt charts.
linkGroup	Link group of tasks for project and scheduling Gantt charts.
outdent	Outdent tasks for project and scheduling Gantt charts.
paste	Paste task for all Gantt charts.
print	Print task for all Gantt charts.
redo	Redo task for all Gantt charts.
separator	Vertical line separating toolbar items for all Gantt charts.
undo	Undo task for all Gantt charts.
undoFilterGroup	Undo /// all Gantt charts.
undoGroup	Undo /// Resource menu item for scheduling Gantt chart.
undoLink	Undo link for /// Task menu item for project and scheduling Gantt charts.

Table 28-5 (Cont.) Valid Values for ToolBarLayout Attributes

Value	Feature Disabled
updateTask	Update Task item for project and scheduling Gantt charts.
zoomIn	Zoom in task for all Gantt charts.
zoomInOutGroup	Zoom in and out tasks ///for all Gantt charts.
zoomOut	Zoom out task for all Gantt charts.
zoomTo	Zoom to Fit item for all Gantt charts.
zoomToGroup	Zoom ///.

[Table 28-6](#) shows the valid values for the `MenuBarLayout` attributes.

Table 28-6 Valid Values for MenuBarLayout Attributes

Value	Feature Disabled
all	Show all standard menu bar items for all Gantt charts.
asList	for all Gantt charts.
asHier	Gantt charts.
collapseAll	Gantt charts.
collapseAllBelow	Gantt charts.
columnsMenu	Gantt charts.
copy	Copy task for all Gantt charts.
createTask	Undo //// all Gantt charts.
cut	Cut task for all Gantt charts.
delete	Delete task for all Gantt charts.
dependencyLines	Gantt charts.
edit	Changes to the data model for all Gantt charts.
editMenu	for all Gantt charts.
expand	Gantt charts.
expandAll	Gantt charts.
expandAllBelow	Gantt charts.
filter	Filter task for project and scheduling Gantt charts.
goToDate	Indent ////task for project and scheduling Gantt charts.
indent	Indent ////.
indentGroup	Undo //// all Gantt charts.
legend	Hide and Show legend toolbar items for all Gantt charts.
link	Link task for project and scheduling Gantt charts.
linkGroup	Link //// for project and scheduling Gantt charts.
listPaneMenu	Undo //// all Gantt charts.

Table 28-6 (Cont.) Valid Values for MenuBarLayout Attributes

Value	Feature Disabled
merge	Undo //// all Gantt charts.
outdent	Outdent tasks for project and scheduling Gantt charts.
paste	Paste task for all Gantt charts.
print	Print task for all Gantt charts.
properties	Gantt charts.
redo	Redo task for all Gantt charts.
resourceMenu	Undo //// all Gantt charts.
separator	Vertical line separating toolbar items for all Gantt charts.
split	Undo //// all Gantt charts.
splitGroup	Undo //// all Gantt charts.
taskMenu	Undo //// all Gantt charts.
timeBucketMenu	Undo //// all Gantt charts.
timeScale	Undo //// all Gantt charts.
undo	Undo task for all Gantt charts.
undoGroup	Undo /// Resource menu item for scheduling Gantt chart.
undoLink	Undo link for /// Task menu item for project and scheduling Gantt charts.
unlink	Update Task item for project and scheduling Gantt charts.
updateTask	Zoom in task for all Gantt charts.
viewMenu	Undo //// all Gantt charts.

Creating Custom Toolbar and Menu Items

You can supply your own menu items and toolbar button by using the `String menuItem` to add custom items to the menu bar, and the `String button` to add custom items to the toolbar. The Gantt chart merges the new menu items with the standard items in the Gantt chart.

The example below shows sample code for specifying a new menu item.

```
<dvt:projectGantt var="task">
<f:facet name="menuBar">
  <af:menu text="My Menu">
    <af:commandMenuItem text="Add.." />
    <af:commandMenuItem text="Create.." />
  </af:menu>
</f:facet>
</dvt:projectGantt>
```

The example below shows sample code for specifying a new toolbar button.

```
<dvt:schedulingGantt var="task">
<f:facet name="toolbar">
  <af:toolbar>
    <af:button text="Custom" disabled="true"/>
  </af:toolbar>
</dvt:schedulingGantt>
```

Actions initiated on the menu bar and toolbar buttons are handled through a registered listener, `DataChangeListener`, on the Gantt chart component. For example, when a user presses the delete button in the toolbar, a `DataChangeEvent` with the ID of the task selected for deletion would be fired on the server. The registered listener is then responsible for performing the actual deletion of the task, and the Gantt chart data model is refreshed with the updated information.

You can register `DataChangeListener` by specifying a method binding using the `dataChangeListener` attribute on the Gantt chart tag. For example, if you put the code in a backing bean in a method called `handleDataChange`, then the setting for the `dataChangeListener` attribute becomes: `"#{myBackingBean.handleDataChange}"`.

The example below shows sample code in a backing bean.

```
public void handleDataChanged(DataChangeEvent evt)
{
  if (DataChangeEvent.DELETE == evt.getActionType())
      .....
}
```

Note:

If your application uses the Fusion technology stack, then you can use data controls to create Gantt charts. By default, a `dataChangeListener` is automatically provided for events. For more information, see the "What You May Need to Know About Data Change Event Handling" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

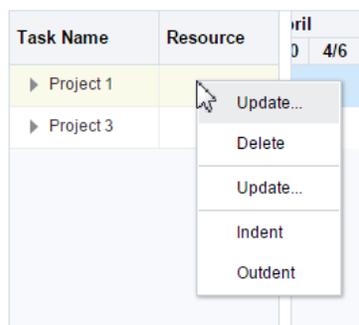
Customizing Gantt Chart Context Menus

When users right-click in the Gantt chart table or chart regions, a context menu is displayed to allow users to perform operations on the Gantt chart. A standard set of options is provided for each region.

You can customize Gantt chart context menus by supplying your own items, and adding, removing, and rearranging items within the menu. Use the `chartPopupMenuLayout` and `tablePopupMenuLayout` attributes of the Gantt chart component to specify the contents and arrangement of context menu items as a set of space-separated Strings.

For example, [Figure 28-23](#) shows a customized table region context menu for a project Gantt chart.

Figure 28-23 Project Gantt Chart Custom Table Region Context Menu



The example below shows sample code for specifying the custom table region context menu in [Figure 28-23](#).

```
<dvt:projectGantt id="gantt1"
value="#{bindings.OrderEmployee1.projectGanttModel}"
dataChangeListener="#{bindings.OrderEmployee1.
projectGanttModel.processDataChanged}"
var="row" startTime="2012-07-01" endTime="2013-05-15"
initiallyExpandAll="true"
summary="Project Gantt chart for shipping orders"
tablePopupMenuLayout="updateTask delete indent outdent"
...
</dvt:projectGantt>
```

[Table 28-7](#) shows the valid values for the `tablePopupMenuLayout` attribute.

Table 28-7 Valid Values for TablePopupMenuLayout Attributes

Value	Feature Disabled
all	Show all standard table region context menu items for all Gantt charts.
clipboard	Cut, Copy, and Paste table region content menu tasks for all Gantt charts.
copy	Copy task for all Gantt charts.
cut	Cut task for all Gantt charts.
delete	Delete task for all Gantt charts.
edit	Changes to the data model for all Gantt charts.
indent	Indent task for project and scheduling Gantt charts.
indentGroup	Indent group of tasks for project and scheduling Gantt charts.
outdent	Outdent tasks for project and scheduling Gantt charts.
paste	Paste task for all Gantt charts.
scrollToTask	Redo task for all Gantt charts.
separator	Vertical line separating toolbar items for all Gantt charts.
updateTask	Update Task item for project and scheduling Gantt charts.

Table 28-8 shows the valid values for the `chartPopupMenuLayout` attribute.

Table 28-8 Valid Values for ChartPopupMenuLayout Attributes

Value	Feature Disabled
all	Show all standard chart region context menu items for all Gantt charts.
clipboard	Cut, Copy, and Paste tasks for all Gantt charts.
copy	Copy task for all Gantt charts.
cut	Cut task for all Gantt charts.
delete	Delete task for all Gantt charts.
edit	Changes to the data model for all Gantt charts.
link	Link task for project and scheduling Gantt charts.
linkGroup	Link group of tasks for project and scheduling Gantt charts.
merge	Outdent tasks for project and scheduling Gantt charts.
paste	Paste task for all Gantt charts.
properties	Print task for all Gantt charts.
separator	Vertical line separating toolbar items for all Gantt charts.
split	Undo task for all Gantt charts.
splitGroup	Undo /// all Gantt charts.
unLink	Undo link for /// Task menu item for project and scheduling Gantt charts.
updateTask	Update Task item for project and scheduling Gantt charts.

You can supply your own menu items using the `tablePopupMenu` and `chartPopupMenu` facets in your Gantt chart. The Gantt chart merges the new menu items with the standard items in the Gantt chart. The example below shows sample code for specifying a custom menu item in the table region context menu.

```
<dvt:projectGantt startTime="#{test.startTime}" endTime="#{test.endTime}"
    value="#{test.treeModel}" var="task">

    <f:facet name="tablePopupMenu">
        <af:popup>
            <af:commandMenuItem text="Custom" disabled="true"/>
        </af:popup>
    </f:facet>
</dvt:projectGantt>
```

You can also dynamically change the context menu at runtime. The example below shows sample code to update a custom context menu on a task bar based on which task is selected in the chart region of a project Gantt chart.

```
<dvt:projectGantt var="task"
    taskSelectionListener="#{backing.handleTaskSelected}">
    <f:facet name="chartPopupMenu">
        <af:popup id="p1" contentDelivery="lazyUncached">
```

```

        <af:menu>
        </af:menu>
    </af:popup>
</f:facet>
</dvt:projectGantt>

```

The `handleTaskSelected` method is specified in a backing bean. The example below shows sample code for the backing bean.

```

public void handleTaskSelected(TaskSelectionEvent evt)
{
    JUCtrlHierNodeBinding _task = (JUCtrlHierNodeBinding)evt.getTask();
    String _type = _task.getAttribute("TaskType");

    RichPopup _popup = m_gantt.getFacet("chartPopupMenu");
    if (_popup != null)
    {
        RichMenu _menu = (RichMenu)_popup.getChildren().get(0);
        _menu.getChildren().clear();
        if ("Summary".equals(_type))
        {
            RichCommandMenuItem _item = new RichCommandMenuItem();
            _item.setId("i1");
            _item.setText("Custom Action 1");
            _menu.getChildren().add(_item);
        }
        else if ("Normal".equals(_type))
        {
            RichCommandMenuItem _item = new RichCommandMenuItem();
            _item.setId("i1");
            _item.setText("Custom Action 2");
            _menu.getChildren().add(_item);
        }
    }
}

```

For more information about using the `af:popup` components see [Using Popup Dialogs, Menus, and Windows](#).

How to Customize the Time Axis of a Gantt Chart

Every Gantt chart is created with a major time axis and a minor time axis. Each time axis has a facet that identifies the level of the axis as major or minor.

The default time axis settings for all Gantt charts are:

- Major time axis: Weeks
- Minor time axis: Days

You can customize the settings of a time axis. However, the setting of a major axis must be a higher time level than the setting of a minor axis. The following values for setting the `scale` on a `dvt:timeAxis` component are listed from highest to lowest:

- `twoyears`
- `year`

- halfyears
- quarters
- twomonths
- months
- twoweeks
- weeks
- days
- sixhours
- threehours
- hours
- halfhours
- quarterhours

The example below shows sample code to set the time axis of a Gantt chart to use months as a major time axis and weeks as the minor time axis.

```
<f:facet name="major">
  <dvt:timeAxis scale="months"/>
</f:facet>
<f:facet name="minor">
  <dvt:timeAxis scale="weeks"/>
</f:facet>
```

The time units you specify for the major and minor axes apply only to the initial display of the Gantt chart. At runtime, the user can zoom in or out on a time axis to display the time unit level at a different level.

You can create a custom time axis for the Gantt chart and specify that axis in the `scale` attribute of `dvt:timeAxis`. The custom time axis will be added to the Time Scale dialog at runtime.

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

You should already have a Gantt chart on your page. If you do not, follow the instructions in this chapter to create a Gantt chart. For information, see [How to Add a Gantt Chart to a Page](#).

To create and use a custom time axis:

1. Implement the `CustomTimescale.java` interface to call the method `getNextDate(Date currentDate)` in a loop to build the time axis. The example below shows sample code for the interface.

```
public interface CustomTimescale
{
    public String getScaleName();
    public Date getPreviousDate(Date ganttStartDate);
    public Date getNextDate(Date currentDate);
    public String getLabel(Date date);
}
```

2. In the Structure window, right-click a Gantt chart node and choose **Go to Properties**.
3. In the **Other** attributes category of the Properties window, for the **CustomTimeScales** attribute, register the implementation of the interface for the custom time axis.

The `customTimeScales` attribute's value is a `java.util.Map` object. The specified map object contains pairs of key/values. The key is the time scale name (`fiveyears`), and the value is the implementation of the `CustomTimeScale.java` interface. For example:

```
customTimeScales="#{project.customTimescales}"
```

4. Also in the Properties window, set the **Scale** attribute for major and minor time axis, and specify the **ZoomOrder** attribute to zoom to the custom times scales. The example below shows sample code for setting a `threeyears` minor time axis and a `fiveyears` major time axis.

```
<f:facet name="major">
    <dvt:timeAxis scale="fiveyears" id="ta1" zoomOrder="fiveyears
threeyears years halfyears quarters months weeks days hours"/>
</f:facet>
<f:facet name="minor">
    <dvt:timeAxis scale="threeyears" id="ta2"/>
</f:facet>
```

Creating and Customizing a Gantt Chart Legend

The optional Gantt chart legend subcomponent includes an area that displays detailed information about the selected task, or metrics about the selected time bucket, and a legend that displays the symbol and color code bar used to represent each type of task in a Gantt chart. At runtime, users can hide or show the information panel using a toolbar button.

The `ganttLegend` tag must be added as a child of the Gantt chart tag in order to provide the legend areas. The content of the legend areas is automatically generated based on the properties for each type of task registered with the `taskbarFormatManager`.

You can customize the information displayed when a task or time bucket is selected by using the `keys` and `label` attributes on the Gantt chart legend tag. The `keys` attribute should specify the data object keys used to retrieve the value to display and the `labels` attribute should contain the corresponding labels for the values retrieved with

the keys. If these attributes are not specified, the legend will use the entire space of the information panel.

You can also add icons to the legend by using the `iconKeys` and `iconLabels` attributes on the Gantt chart legend tag. Icons will be automatically resized to 12 by 12 pixels if the icon size is too large.

The example below shows sample code to display information about an On Hold task in the legend of a project Gantt chart.

```
<dvt:projectGantt var="task">
  <dvt:ganttLegend id="gl" keys="TaskName StartTime EndTime" labels="Name
Start Finish" icons="images/wait.png" iconLabels="OnHold"/>
</dvt:projectGantt>
```

How to Specify Custom Data Filters

You can change the display of data in a Gantt chart using a data filter dropdown list on the toolbar. Gantt charts manage all predefined and user-specified data filters using a `FilterManager`.

Filter objects contain information including:

- A unique ID for the filter
- The label to display for the filter in the dropdown list
- An optional JavaScript method to invoke when the filter is selected

You can define your own filter by creating a filter object and then registering the object using the `addFilter` method on the `FilterManager`. The example below shows sample code for registering a Resource filter object with the `FilterManager`.

```
FilterManager _manager = m_gantt.getFilterManager();

// ID for filter display label  javascript callback (optional)
_manager.addFilter((new Filter(RESOURCE_FILTER, "Resource...",
"showResourceDialog")));
```

When the user selects a filter, a `FilterEvent` is sent to the registered `FilterListener` responsible for performing the filter logic. The `filterListener` attribute on the Gantt chart component is used to register the listener. When implemented by the application, the data model is updated and the Gantt chart component displays the filtered result. The example below shows sample code for a `FilterListener`.

```
public void handleFilter(FilterEvent event)
{
    String _type = event.getType();
    if (FilterEvent.ALL_TASKS.equals(_type))
    {
        // update the gantt model as appropriate
    }
}
```

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

You should already have a Gantt chart on your page. If you do not, follow the instructions in this chapter to create a Gantt chart. For information, see [How to Add a Gantt Chart to a Page](#).

To specify a custom data filter:

1. In the Structure window, right-click the Gantt chart component and choose **Go to Properties**.
2. In the Behavior category of the Properties window, in the `FilterListener` field, enter a method reference to the `FilterListener` you defined. For example, `"#{project.handleFilter}"`.

Specifying Nonworking Days in a Gantt Chart

You can specify nonworking days in a Gantt chart. By default, nonworking days are shaded gray, but you can select a custom color to be used for nonworking days.

How to Specify Weekdays as Nonworking Days

If certain weekdays are always nonworking days, then you can indicate the days of the week that fall in this category.

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

You should already have a Gantt chart on your page. If you do not, follow the instructions in this chapter to create a Gantt chart. For information, see [How to Add a Gantt Chart to a Page](#).

To identify weekdays as nonworking days:

1. In the Structure window, right-click the Gantt chart component and choose **Go to Properties**.
2. In the Appearance category of the Properties window, in the `NonWorkingDaysOfWeek` field, enter the string of days that you want to identify as nonworking days for each week. For example, to specify that Saturday and Sunday are nonworking days, enter the following string: `"sat sun"`.

Alternatively, you can create a method in a backing bean to programmatically identify the nonworking days. For example, if you put the code in a backing bean in a method called `getNonWorkingDaysOfWeek`, then the setting for the

`nonWorkingDaysOfWeek` attribute becomes: `"#{myBackingBean.nonWorkingDays}"`.
The example below shows sample code in a backing bean.

```
public int[] getNonWorkingDaysOfWeek(){ if (locale == Locale.EN_US
    return new int[] {Calendar.SATURDAY, Calendar.SUNDAY}; else      ....}
```

3. Optionally, specify a custom color in the `NonWorkingDaysColor` field. The value you enter for this attribute must be a hexadecimal color string.

How to Identify Specific Dates as Nonworking Days

You can enter specific dates as nonworking days in a Gantt chart when individual weekdays are not sufficient.

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

You should already have a Gantt chart on your page. If you do not, follow the instructions in this chapter to create a Gantt chart. For information, see [How to Add a Gantt Chart to a Page](#).

To identify specific dates as nonworking days:

1. In the Structure Window, right-click the Gantt chart component and choose **Go to Properties**.
2. In the Properties window, select the **Appearance** attributes category.
3. In the `nonWorkingDays` field, enter the string of dates that you want to identify as nonworking days. For example: `"2008-07-04 2008-11-28 2008-12-25"`.

Alternatively, for more flexibility, you can create a method in a backing bean to programmatically identify the nonworking days. For example, if you put the code in a backing bean in a method called `getNonWorkingDays`, then the setting for the `nonWorkingDays` attribute becomes: `"#{myBackingBean.nonWorkingDays}"`.

4. Optionally, specify a custom color in the `nonWorkingDaysColor` field. The value you enter for this attribute must be a hexadecimal color string.

How to Apply Read-Only Values to Gantt Chart Features

User interactions with a Gantt chart can be customized to disable features by setting the `featuresOff` property to specify read-only values.

[Table 28-9](#) shows the valid values and the disabled feature for the Gantt chart types.

Table 28-9 Valid Values for Read-Only Attributes

Value	Feature Disabled
<code>asListMenu</code>	Show as List menu item for all Gantt charts.
<code>asHierMenu</code>	Show as Hierarchy menu item for all Gantt charts/

Table 28-9 (Cont.) Valid Values for Read-Only Attributes

Value	Feature Disabled
clipboard	Cut, Copy, and Paste tasks for all Gantt charts.
clipboardMenu	Cut, Copy, and Paste menu items for all Gantt charts.
clipboardToolbar	Cut, Copy, and Paste toolbar items for all Gantt charts.
clipboardRightMenu	Cut, Copy, and Paste right menu items for all Gantt charts.
collapseAllBelowMenu	Collapse All Below menu item for all Gantt charts.
collapseAllMenu	Collapse All menu item for all Gantt charts.
columnsMenu	Columns menu item for all Gantt charts.
createResourceMenu	Create Resource menu item for resource utilization and scheduling Gantt charts.
createResourceMT	Create Resource menu and toolbar items for resource utilization and scheduling Gantt charts.
createResourceToolbar	Create Resource toolbar item for resource utilization and scheduling Gantt charts.
createTaskMenu	Create Task menu for project and scheduling Gantt charts.
createTaskMT	Create Task menu and toolbar for project and scheduling Gantt charts.
createTaskToolbar	Create Task toolbar for project and scheduling Gantt charts.
deleteMenu	Delete menu item for project and scheduling Gantt charts.
deleteMenus	Delete menu, right menu, and toolbar items for project and scheduling Gantt charts.
deleteRightMenu	Delete right menu item for project and scheduling Gantt charts.
deleteToolbar	Delete toolbar item for project and scheduling Gantt charts.
dependencyLines	Show and Hide dependency lines for project and scheduling Gantt charts. This includes the dependency menu option for project Gantt charts.
edit	Changes to the data model for all Gantt charts.
editMenu	Edit menu item for all Gantt charts.
expandAllBelowMenu	Expand All Below menu item for all Gantt charts.
expandAllMenu	Expand All menu item for all Gantt charts.
expandMenu	Expand menu item for all Gantt charts.
filter	Hide the data filter operation on the toolbar for all Gantt charts.
goToDateMenu	Go to Date menu item for all Gantt charts.
indenting	Indent and Outdent tasks for project and scheduling Gantt charts.
indentingMenu	Indent and Outdent menu items: Task for project, and Resource for scheduling and resource utilization resource Gantt charts.
indentingMenus	Indent and Outdent menu and toolbar and right menu items for all Gantt charts.
indentingRightMenu	Indent and Outdent right menu items for all Gantt charts.

Table 28-9 (Cont.) Valid Values for Read-Only Attributes

Value	Feature Disabled
indentingToolbar	Indent and Outdent toolbar items for all Gantt charts.
legend	Hide and Show legend and task information for all Gantt charts.
legendMenu	Hide and Show legend menu items for all Gantt charts.
legendToolbar	Hide and Show legend toolbar items for all Gantt charts.
linking	Link and Unlink tasks for project and scheduling Gantt charts.
linkingMenu	Link and Unlink menu items for project and scheduling Gantt charts.
linkingMenus	Link and Unlink menu, right menu, and toolbar items for project and scheduling Gantt charts.
linkingRightMenu	Link and Unlink right menu items for project and scheduling Gantt charts.
linkingToolbar	Link and Unlink toolbar items for project and scheduling Gantt charts.
listPaneMenu	List Pane menu item for all Gantt charts.
print	Print task for all Gantt charts.
printMenu	Print menu item for all Gantt charts.
printToolbar	Print toolbar item for all Gantt charts.
properties	Show property dialogs for all Gantt charts.
propertiesMenu	Properties menu item for all Gantt charts.
propertiesRightMenu	Properties right menu item for all Gantt charts.
resourceMenu	Resource menu item for resource utilization and scheduling Gantt charts.
snapToMenu	Snap To menu item scheduling Gantt chart.
snapToRightMenu	Snap To right menu item for scheduling Gantt chart.
split	Split and Merge tasks for project Gantt chart.
splittingMenu	Split and Merge menu items for project Gantt chart.
splittingMenus	Split and Merge menu, right menu, and toolbar items for project Gantt chart.
splittingRightMenu	Split and Merge right menu items for project Gantt chart.
splittingToolbar	Split and Merge toolbar items for project Gantt chart.
taskMenu	Task menu for project and scheduling Gantt charts.
timeAxisMenu	Time Axis menu item for all Gantt charts.
timeBucketMenu	Time Bucket menu item for resource utilization Gantt chart.
undo	Undo and redo tasks for all Gantt charts.
undoMenu	Undo and Redo menu items for all Gantt charts.
updateResourceMenu	Update Resource menu item for scheduling Gantt chart.
updateTaskMenu	Update Task menu item for project and scheduling Gantt charts.

Table 28-9 (Cont.) Valid Values for Read-Only Attributes

Value	Feature Disabled
updateTaskMT	Update Task Edit item, Update Task toolbar item, and right menu items for project and scheduling Gantt charts.
updateTaskRightMenu	Update Task right menu item for project and scheduling Gantt charts.
updateTaskToolbar	Update Task toolbar item for project and scheduling Gantt charts.
undoToolbar	Undo and Redo toolbar items for all Gantt charts.
view	Show as list, Show as hierarchy, Columns, Expand and Collapse tasks for all Gantt charts.
viewMenu	View menu items for all Gantt charts.
zoom	Changes to the zoom level for all Gantt charts.
zoomToolbar	Zoom menu item for all Gantt charts.
zoomToToolbar	Zoom to Fit menu toolbar item for all Gantt charts.

Before you begin:

It may be helpful to have an understanding of how Gantt chart attributes and Gantt chart child components can affect functionality. For more information, see [Configuring Gantt Charts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Gantt Chart Components](#).

You should already have a Gantt chart on your page. If you do not, follow the instructions in this chapter to create a Gantt chart. For information, see [How to Add a Gantt Chart to a Page](#).

To set read-only values on Gantt chart features:

1. In the Structure window, right-click the Gantt chart node and choose **Go to Properties**.
2. In the **Behavior** attributes category of the Properties window, for the `featuresOff` attribute, enter one or more String values to specify the Gantt chart features to disable.

For example, to disable user interactions for editing the data model, printing, or changing the zoom level of a Gantt chart, use the following setting for the `featuresOff` attribute: `edit print zoom`

Alternatively, you can create a method in a backing bean to programmatically identify the features to be disabled. For example, if you put the code in a backing bean in a method called `whatToTurnOff` that returns a String array of the values, then the setting for the `featuresOff` attribute becomes:

```
"#{BackingBean.whatToTurnOff}"
```

What You May Need to Know About Skinning and Customizing the Appearance of Gantt Charts

For the complete list of Gantt chart skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactive Features to Gantt Charts

ADF DVT Gantt Charts support multiple interactive features, including adding a page control as an alternative to a scrollbar, synchronized scrolling, adding a double-click event to a task bar, and printing Gantt charts.

Performing an Action on Selected Tasks or Resources

A Gantt chart allows users to select one or more rows in the table region of a Gantt chart representing tasks or resources and perform some actions on those rows. When the selection state of a Gantt chart changes, the Gantt chart triggers selection events. A `selectionEvent` event reports which rows were just deselected and which rows were just selected.

To listen for selection events on a Gantt chart, you can register a listener on the Gantt chart either using the `selectionListener` attribute or by adding a listener to the Gantt chart using the `addselectionListener()` method. The listener can then access the selected rows and perform some actions on them.

The current selection, that is the selected row or rows, are the `RowKeySet` object, which you obtain by calling the `getSelectedRowKeys()` method for the Gantt chart. To change a selection programmatically, you can do either of the following:

- Add `rowKey` objects to, or remove `rowKey` objects from, the `RowKeySet` object.
- Make a particular row current by calling the `setRowIndex()` or the `setRowKey()` method on the Gantt chart. You can then either add that row to the selection, or remove it from the selection, by calling the `add()` or `remove()` method on the `RowKeySet` object.

The example below shows a portion of a Gantt chart in which a user can select some rows then click the **Delete** button to delete those rows. Note that the actions listener is bound to the `performDelete` method on the `mybean` managed bean.

```
<f:facet name="nodeStamp">
  <af:column headerText= ...>
    <af:outputText binding="{mybean.table}" value= ... />
  </af:column>
  ...
</f:facet>
<af:button text="Delete" actionListener="{mybean.performDelete}"/>
```

The example below shows an actions method, `performDelete`, which iterates through all the selected rows and calls the `markForDeletion` method on each one.

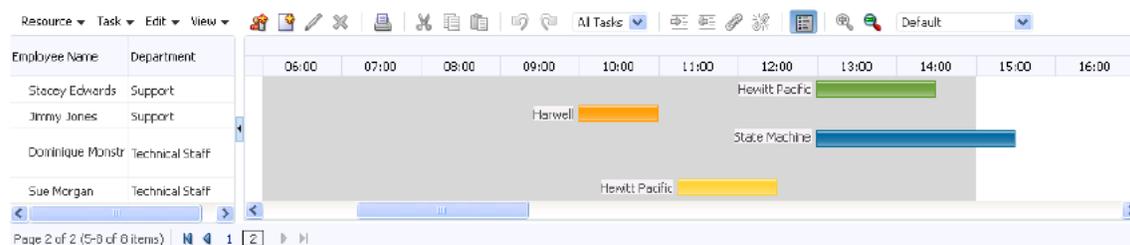
```
public void performDelete(ActionEvent action)
{
    UIXTable table = getTable();
    Iterator selection = table.getSelectedRowKeys().iterator();
    Object oldKey = table.getRowKey();
    try
    {
        while(selection.hasNext())
        {
            Object rowKey = selection.next();
            table.setRowKey(rowKey);
            MyRowImpl row = (MyRowImpl) table.getRowData();
            //custom method exposed on an implementation of Row interface.
            row.markForDeletion();
        }
    }
    finally
    {
        // restore the old key:
        table.setRowKey(oldKey);
    }
}
```

Using Page Controls for a Gantt Chart

For Gantt chart table regions, you can use a page control as an alternative to horizontal scrolling for both desktop applications and for mobile browsers on touch devices. This control is only available when there are more rows than the data fetch size, and the component is not being stretched by its containing layout component.

The page control displays as a footer to the table region as shown in [Figure 28-24](#).

Figure 28-24 Gantt Chart Page Control



When you are developing an ADF Faces web application, by default Gantt chart table regions use a horizontal scroll bar for displaying columns over the size of the data being fetched. To configure a page control for desktop devices, for the `schedulingGantt`, `projectGantt`, or `resourceUtilizationGantt` component table region, set the `scrollPolicy` attribute to `page`.

While a standard ADF Faces web application will run in mobile device browsers, because the user interaction is different and because screen size is limited, when your

application needs to run on a mobile device, you should create touch device-specific versions of the pages. For more information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).

By default, when rendered on mobile devices, Gantt chart table regions display a page control that allows the user to jump to specific pages of rows. For all Gantt charts to display on a mobile device, you should:

- Place the Gantt chart component within a flowing container (that is, a component that does not stretch its children). For more information about flowing container components, see [Geometry Management and Component Stretching](#).
- Leave the `scrollPolicy` attribute set to `auto` (default for this setting on mobile devices is paginated display of the pivot table).

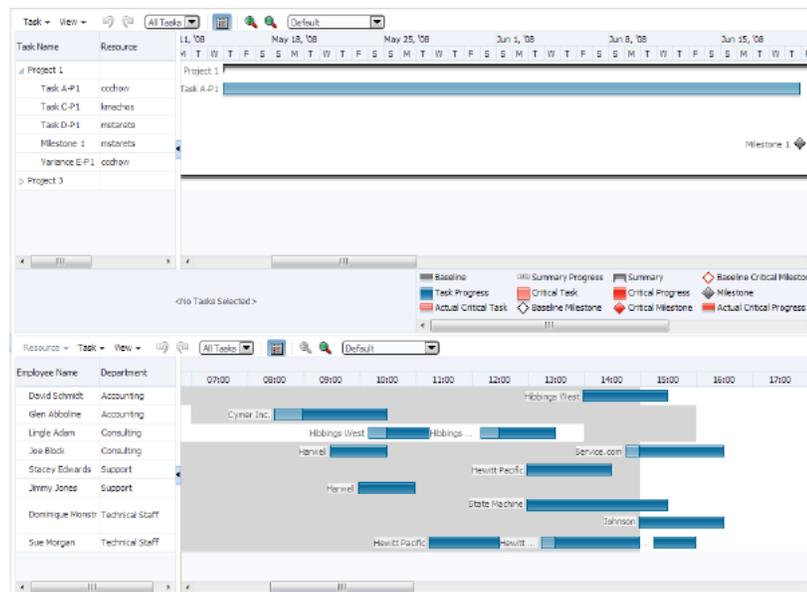
If the Gantt chart is not in a flowing container, or if those attributes are not set correctly, the table region will display a scroll bar instead of pages.

Configuring Synchronized Scrolling Between Gantt Charts

You can configure synchronized horizontal scrolling between the chart side of two Gantt charts by using an `af:clientListener` component.

For example, you may wish to synchronize the scroll bars of a project Gantt chart and a resource utilization Gantt chart to view tasks and resources for the same project as illustrated in [Figure 28-25](#).

Figure 28-25 Synchronized Scrolling Between Gantt Charts



To configure synchronized scrolling between Gantt charts, use an `af:clientListener` component to listen for the `chartHorizontalScroll` event on the chart side of the Gantt chart being scrolled and set the scroll position on the other Gantt chart.

In this example, inline JavaScript is used to define the methods for synchronized scrolling within an `af:resource` tag. The example below shows the code for the

synchronized scrolling methods. For more information, see [Adding JavaScript to a Page](#).

```

<af:resource type="javascript">
  var gantt1ScrollStart = null;
  var gantt2ScrollStart = null;
  //called when the top gantt component is scrolled
  function handleTopScroll(event)
  {
    var eventScrollStart = event.getScrollStart();
    if (gantt2ScrollStart == null || gantt2ScrollStart ==
eventScrollStart)
    {
      // clear synced gantt scroll start
      gantt2ScrollStart = null;
      // find the corresponding gantt component
      var gantt = AdfPage.PAGE.findComponent("demo:gantt2");
      if (gantt1ScrollStart != event.getScrollStart())
      {
        // set the scrollStart position of the synced gantt to match
the
        event's value
        var scrollStart = gantt.getChartScrollStart();
        if (scrollStart != eventScrollStart)
        {
          //save the scrollStart value to stop feedback
          gantt1ScrollStart = eventScrollStart;
          gantt.setChartScrollStart(eventScrollStart);
        }
      }
    }
    event.cancel();
  }

  // called when the bottom gantt component is scrolled
  function handleBottomScroll(event)
  {
    var eventScrollStart = event.getScrollStart();
    if (gantt1ScrollStart == null || gantt1ScrollStart ==
eventScrollStart)
    {
      // clear synced gantt scroll start
      gantt1ScrollStart = null;
      // find the corresponding gantt component
      var gantt = AdfPage.PAGE.findComponent("demo:gantt1");
      if (gantt2ScrollStart != event.getScrollStart())
      {
        // set the scrollStart position of the synced gantt to match
the
        event's value
        var scrollStart = gantt.getChartScrollStart();
        if (scrollStart != eventScrollStart)
        {
          //save the scrollStart value to stop feedback
          gantt2ScrollStart = eventScrollStart;

```

```

        gantt.setChartScrollStart(eventScrollStart);
    }
}
}
event.cancel();
}
</af:resource

```

The example below shows the code in both Gantt charts to specify a `clientListener` to listen for the Gantt charts' `scrollEvent` of type `chartHorizontalScroll` and invoke the `handleTopScroll` and `handleBottomScroll` methods defined in the `af:resource` component in the above example.

```

<dvt:projectGantt id="ganttl" var="task" startTime="2008-04-22"
    endTime="2008-09-31" inlineStyle="height:400px;"
    value="#{projectGantt.model}"
    tooltipKeys="#{projectGantt.tooltipKeys}"
    tooltipKeyLabels="#{projectGantt.tooltipLabels}"
    summary="Project Gantt">
    <f:facet name="major">
        <dvt:timeAxis scale="weeks" id="ta1"/>
    </f:facet>
    <f:facet name="minor">
        <dvt:timeAxis scale="days" id="ta2"/>
    </f:facet>
    <f:facet name="nodeStamp">
        <af:column headerText="Task Name" id="c1">
            <af:outputText value="#{task.taskName}" id="ot1"/>
        </af:column>
    </f:facet>
        <af:column headerText="Resource" id="c2">
            <af:outputText value="#{task.resourceName}" id="ot2"/>
        </af:column>
        <af:column headerText="Start Date" id="c3">
            <af:outputText value="#{task.startTime}" id="ot3"/>
        </af:column>
        <af:column headerText="End Date" id="c4">
            <af:outputText value="#{task.endTime}" id="ot4"/>
        </af:column>
    <dvt:ganttLegend keys="#{projectGantt.legendKeys}"
        labels="#{projectGantt.legendLabels}"
        id="gl1"/>
    <af:clientListener type="chartHorizontalScroll"
method="handleTopScroll"/>
</dvt:projectGantt>
<dvt:schedulingGantt id="gantt2" startTime="2006-12-21 01:00"
    endTime="2006-12-22 23:00"
    value="#{schedulingGantt.model}" var="resourceObj"
    inlineStyle="height:400px;"
    tooltipKeyLabels="#{schedulingGantt.tooltipLabels}"
    tooltipKeys="#{schedulingGantt.tooltipKeys}"
    summary="Scheduling Gantt Demo">
    <f:facet name="major">
        <dvt:timeAxis scale="days" id="ta3"/>
    </f:facet>

```

```

</f:facet>
<f:facet name="minor">
  <dvt:timeAxis scale="hours" id="ta4"/>
</f:facet>
<f:facet name="nodeStamp">
  <af:column headerText="Employee Name" id="c5">
    <af:outputText value="#{resourceObj.resourceName}" id="ot5"/>
  </af:column>
</f:facet>
<af:column headerText="Department" id="c6">
  <af:outputText value="#{resourceObj.department}" id="ot6"/>
</af:column>
<dvt:ganttLegend keys="#{schedulingGantt.legendKeys}"
  labels="#{schedulingGantt.legendLabels}"
  id="gl2"/>
  <af:clientListener type="chartHorizontalScroll"
method="handleBottomScroll"/>
</dvt:schedulingGantt>

```

Printing a Gantt Chart

The ADF Gantt chart provides a helper class (`GanttPrinter`) that can generate a Formatted Object (FO) for use with Apache or XML Publisher to produce PDF files.

Print Options

In general, the `GanttPrinter` class prints the Gantt chart content as it appears on your screen. For example, if you hide the legend in the Gantt chart, then the legend will not be printed. Similarly, if you deselect a column in the List Pane section of the View Menu, then that column will not be visible in the Gantt chart and will not appear in the printed copy unless you take advantage of the column visibility print option.

You can use the following print options in the `GanttPrinter` class:

- **Column visibility:** The `setColumnVisible` method lets you control whether individual columns in the list region of the Gantt chart will appear in the printed output.

For example, to hide the first column in the list region of a Gantt chart, use the following code, where the first parameter of the method is the zero-based index of the column and the second parameter indicates if the column should be visible in the printed Gantt chart: `_printer.setColumnVisible(o, false);`

- **Margins:** The `setMargin` method of the `GanttPrinter` lets you specify the top, bottom, left, and right margins in pixels as shown in the following code, where `_printer` is an instance of the `GanttPrinter` class:

```
_printer.setMargin(25, 16, 66, 66);
```

- **Page size:** The `setPageSize` method of the `GanttPrinter` class lets you specify the height and width of the printed page in pixels as shown in the following code, where `_printer` is an instance of the `GanttPrinter` class:

```
_printer.setPageSize (440, 600);
```

- **Time period:** The `setStartTime` and `setEndTime` methods of the `GanttPrinter` class let you identify the time period of the Gantt chart that you want to print.

The example below shows sample code for setting a specific time period in the Gantt chart for printing, where `startDate` and `endDate` are variables that represent the desired dates and `_printer` is an instance of the `GanttPrinter` class.

```
_printer.setStartTime(startDate);  
_printer.setEndTime(endDate);
```

Action Listener to Handle the Print Event

The Gantt chart toolbar includes a print button that initiates a print action. To print a Gantt chart, you must create an `ActionListener` to handle the print event.

The code in the `ActionListener` should include the following processes:

1. Access the servlet's output stream.
2. Generate the FO. This process includes creating an instance of the `GanttPrinter` class and entering the code for any print options that you want to use.
3. Generate the PDF.

The example below shows the code for an `ActionListener` that handles the print event. This listener includes settings for all the print options available in the `GanttPrinter` helper class.

```
public void handleAction(GanttActionEvent evt)  
{  
    if (GanttActionEvent.PRINT == evt.getActionType())  
    {  
        FacesContext _context = FacesContext.getCurrentInstance();  
        ServletResponse _response = (ServletResponse)  
            _context.getExternalContext().getResponse();  
        _response.setContentType("application/pdf");  
        ServletOutputStream _sos = _response.getOutputStream();  
        // Generate FO.  
        GanttPrinter _printer = new GanttPrinter(m_gantt);  
        // Set column visibility by column index.  
        _printer.setColumnVisible(0, false);  
        // Set start and end date.  
        _printer.setStartTime(startDate);  
        _printer.setEndTime(endDate);  
        // Set top, bottom, left, and right margins in pixels.  
        _printer.setMargin(25, 16, 66, 66);  
        // Set height and width in pixels.  
        _printer.setPageSize(440, 660);  
        File _file = File.createTempFile("gantt", "fo");  
        OutputStream _out = new FileOutputStream(_file);  
        _printer.print(_out);  
        _out.close();  
        // generate PDF.  
        FOProcessor _processor = new FOProcessor();  
        _processor.setData(new FileInputStream(_file), "UTF-8");  
        _processor.setOutputFormat(FOProcessor.FORMAT_PDF);  
        _processor.setOutput(_sos);  
        _processor.generate();  
        _context.responseComplete();  
        response.setHeader("Cache-Control", "no-cache");  
    }  
}
```

```

    }
}

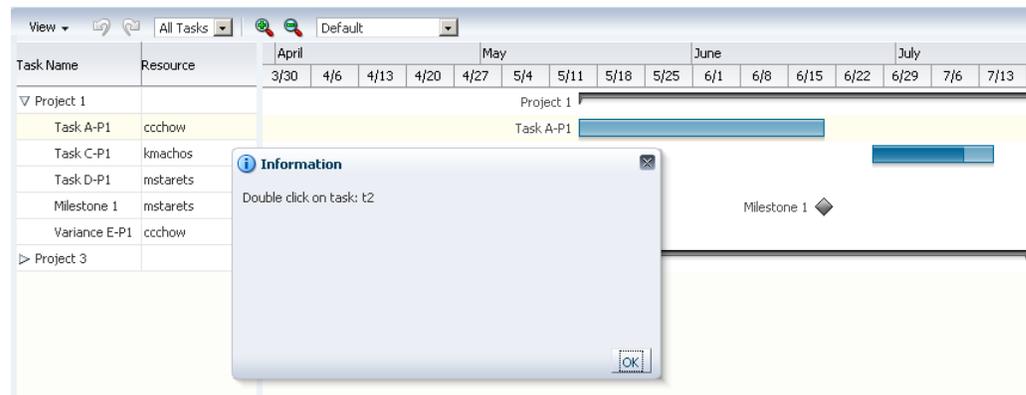
```

Adding a Double-Click Event to a Task Bar

Gantt chart components support a double-click event on a task bar. For example, you may want to display detailed information about a task in a popup window.

Figure 28-26 shows a project Gantt chart with a double-click event on a task bar.

Figure 28-26 Task Bar with Double-Click Event



The example below show sample code for adding a double-click event to a task bar.

```

<dvt:projectGantt id="projectGanttDoubleClick"
  startTime="2008-04-01" endTime="2008-09-30"
  value="#{projectGanttDoubleClick.model}"
  var="task"
  doubleClickListener="#{projectGanttDoubleClick.handleDoubleClick}">
</dvt:projectGantt>

```

Implement the `handleDoubleClick` method in a backing bean, for example:

```
public void handleDoubleClick(DoubleClick event)
```

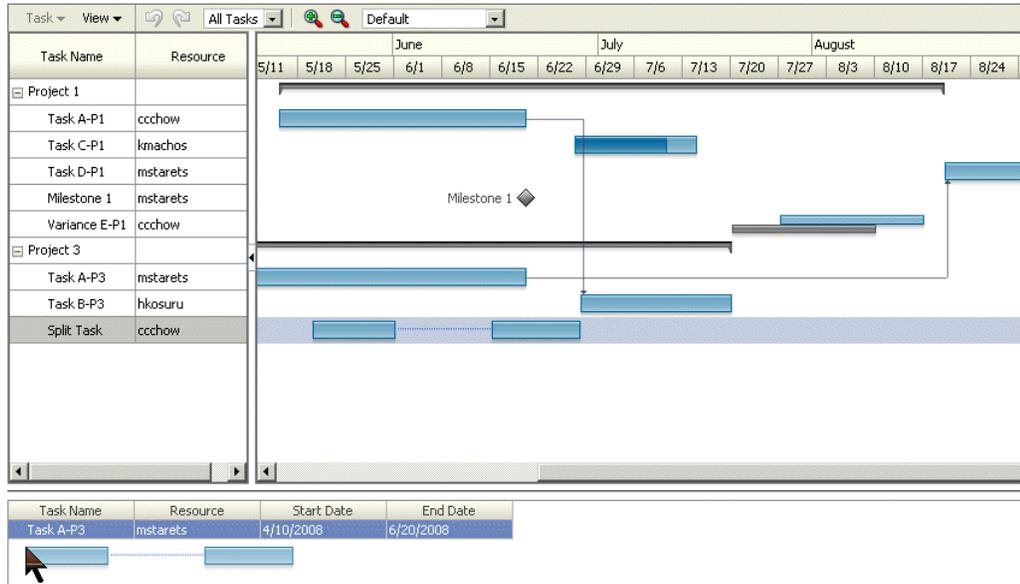
Using Gantt Charts as a Drop Target or Drag Source

You can add drag and drop functionality that allows users to drag an item from a collection, for example, a row from a table, and drop it into another collection component, such as a tree. Project and scheduling Gantt chart components can be enabled as drag sources as well as drop targets for ADF table or tree table components. A resource utilization Gantt chart component can be enabled only as a drop target.

The application must register the Gantt chart component as a drag source or drop target by adding the `af:collectionDragSource` or `af:collectionDropTarget` behavior tags respectively as a child to the Gantt tag. For example, you can use the `af:collectionDragSource` to register a drop listener that would be invoked when a project Gantt chart task is dragged from a table region onto a separate table.

Figure 28-27 shows a project Gantt chart with tasks dragged from the table region onto a table of tasks.

Figure 28-27 Project Gantt Chart as Drag Source



The example below shows sample code for adding drag and drop functionality to a project Gantt chart.

```
<dvt:projectGantt id="projectGanttDragSource"
    startTime="2008-04-01" endTime="2008-09-30"
    value="#{projectGanttDragSource.model}"
    var="task"
    summary="Project Gantt Drag Source Demo">
  <f:facet name="major">
    <dvt:timeAxis scale="months" id="ta1"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="weeks" id="ta2"/>
  </f:facet>
  <af:collectionDragSource actions="COPY MOVE" modelName="treeModel"/>
  <f:facet name="nodeStamp">
    <af:column headerText="Task Name" id="c1">
      <af:outputText value="#{task.taskName}" id="ot1"/>
    </af:column>
  </f:facet>
    <af:column headerText="Resource" id="c2">
      <af:outputText value="#{task.resourceName}" id="ot2"/>
    </af:column>
    <af:column headerText="Start Date" id="c3">
      <af:outputText value="#{task.startTime}" id="ot3"/>
    </af:column>
    <af:column headerText="End Date" id="c4">
      <af:outputText value="#{task.endTime}" id="ot4"/>
    </af:column>
  </f:facet>
</dvt:projectGantt>
```

```
</af:column>  
</dvt:projectGantt>
```

The example below shows sample code for the listener method for handling the drop event.

```
public DnDAction onTableDrop(DropEvent evt)  
{  
    Transferable _transferable = evt.getTransferable();  
  
    // Get the drag source, which is a row key, to identify which row has  
    // been dragged.  
    RowKeySetImpl _rowKey =  
(RowKeySetImpl)_transferable.getTransferData(DataFlavor.ROW_KEY_SET_FLAVOR)  
.getData();  
  
    // Set the row key on the table model (source) to get the data.  
    // m_tableModel is the model for the Table (the drag source).  
    object _key = _rowKey.iterator().next();  
    m_tableModel.setRowKey(_key);  
  
    // See on which resource this is dropped (specific for scheduling  
    // Gantt chart).  
    String _resourceId = _transferable.getData(String.class);  
    Resource _resource = findResourceById(_resourceId);  
  
    // See on what time slot did this dropped.  
    Date _date = _transferable.getData(Date.class);  
  
    // Add code to update your model here.  
  
    // Refresh the table and the Gantt chart.  
  
    RequestContext.getCurrentInstance().addPartialTarget(_evt.getDragComponent(  
    ));  
    RequestContext.getCurrentInstance().addPartialTarget(m_gantt);  
  
    // Indicate the drop is successful.  
    return DnDAction.COPY;  
}
```

For a detailed procedure about adding drag and drop functionality for collections, see [Adding Drag and Drop Functionality for Collections](#).

Using Timeline Components

This chapter describes how to use the ADF Data Visualization `timeline` component to display data using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the component.

If your application uses the Fusion technology stack, then you can also use data controls to create timelines. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About Timeline Components](#)
- [Using Timeline Components](#)
- [Adding Data to Timeline Components](#)
- [Customizing Timeline Display Elements](#)
- [Adding Interactive Features to Timelines](#)

About Timeline Components

An ADF DVT Timeline is an interactive data visualization tool that allows users to view events in chronological order and easily navigate forwards and backwards within a defined time range. Events are represented as timeline items using simple ADF components to display information such as text and images, or supply actions such as links.

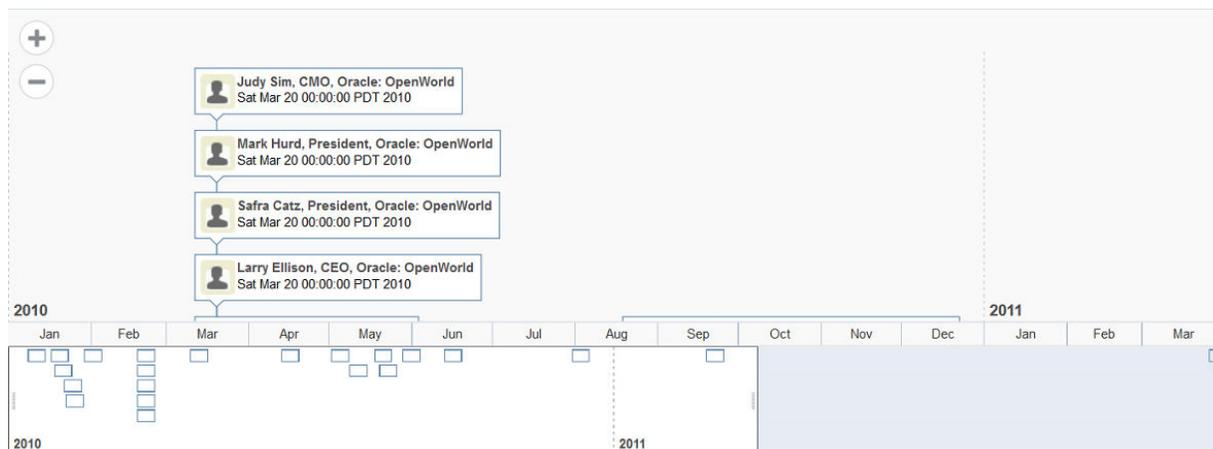
A dual timeline can be configured to display two series of events to allow a side-by-side comparison of related information. The timeline component also supports an adjustable time range to change the view for zooming in or out.

Timeline Use Cases and Examples

A timeline is composed of the display of events as timeline items along a time axis, a movable overview window that corresponds to the period of viewable time in the timeline, and an overview time axis that displays the total time increment for the timeline. A horizontal zoom control is available to change the viewable time range. Timeline items corresponding to events display related information or actions and are represented by a line feeler to the time axis and a marker in the overview time axis.

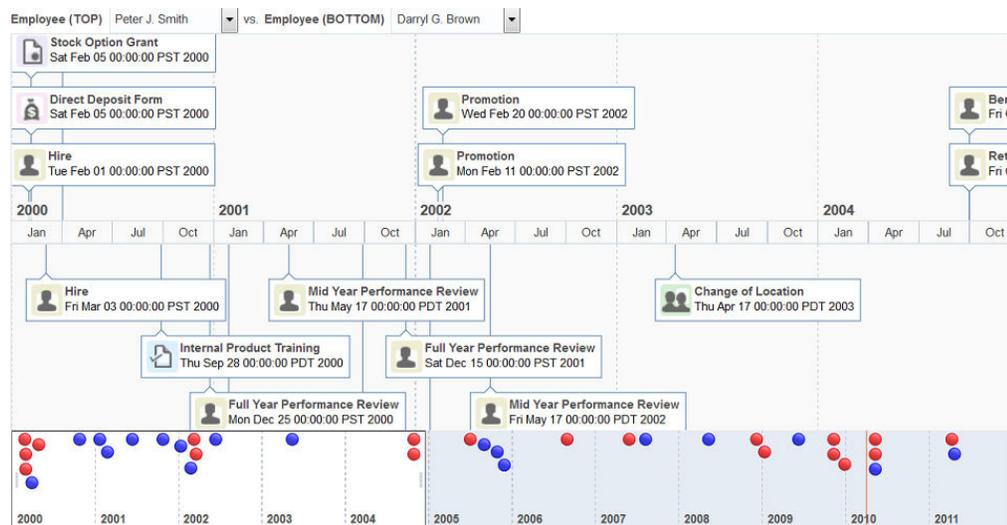
For example, the timeline in [Figure 29-1](#) is configured to display the chronological order of Employee Presentations in the ADF Faces Components Demo application. In this example, timeline items representing each event display information about the event with a label and a subheading. The overview window defines the time range for the display of the timeline items, adjustable by changing the zoom control or by changing the edges of the window to a larger or smaller size. When selection is configured, the timeline item, line feeler, and the event marker in the overview panel are highlighted.

Figure 29-1 Timeline of Employee Hire Dates



A dual timeline can be used for comparison of up to two series of events. [Figure 29-2](#) illustrates a dual timeline comparing employee change events for two employees over a ten year time period. Timeline events are displayed using a quarterly year time axis within the three plus year overview window. The current date is represented with a line in the overview time axis.

Figure 29-2 Dual Timeline Comparing Employee Change Events



End User and Presentation Features

To understand how timelines are used and can be customized, it is helpful to understand these elements and features.

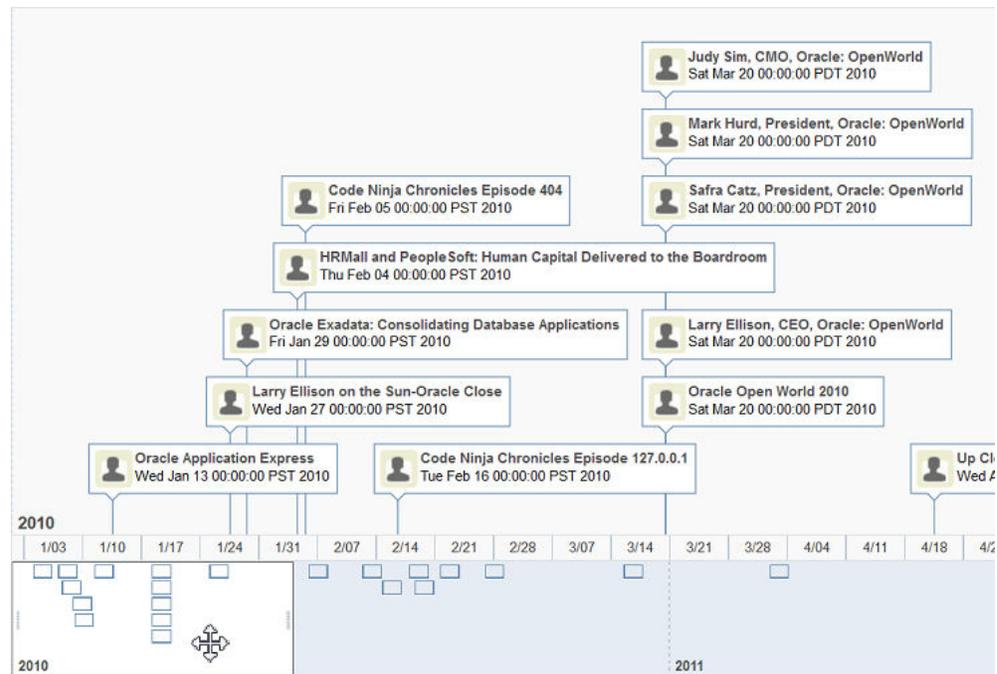
Timeline Overview Options

By default timelines support an overview panel that includes a movable overview window corresponding to the period of viewable time in the timeline. An optional

overview time axis that displays the total time increment for the timeline with adjustable zoom levels can be added to a timeline.

Figure 29-3 shows a timeline of speaking engagements with the overview panel displaying events occurring during a specific time period of the timeline. The time axis for the timeline displays the total time increment with adjustable zoom levels between hourly and half years.

Figure 29-3 Timeline Overview Panel and Time Axis

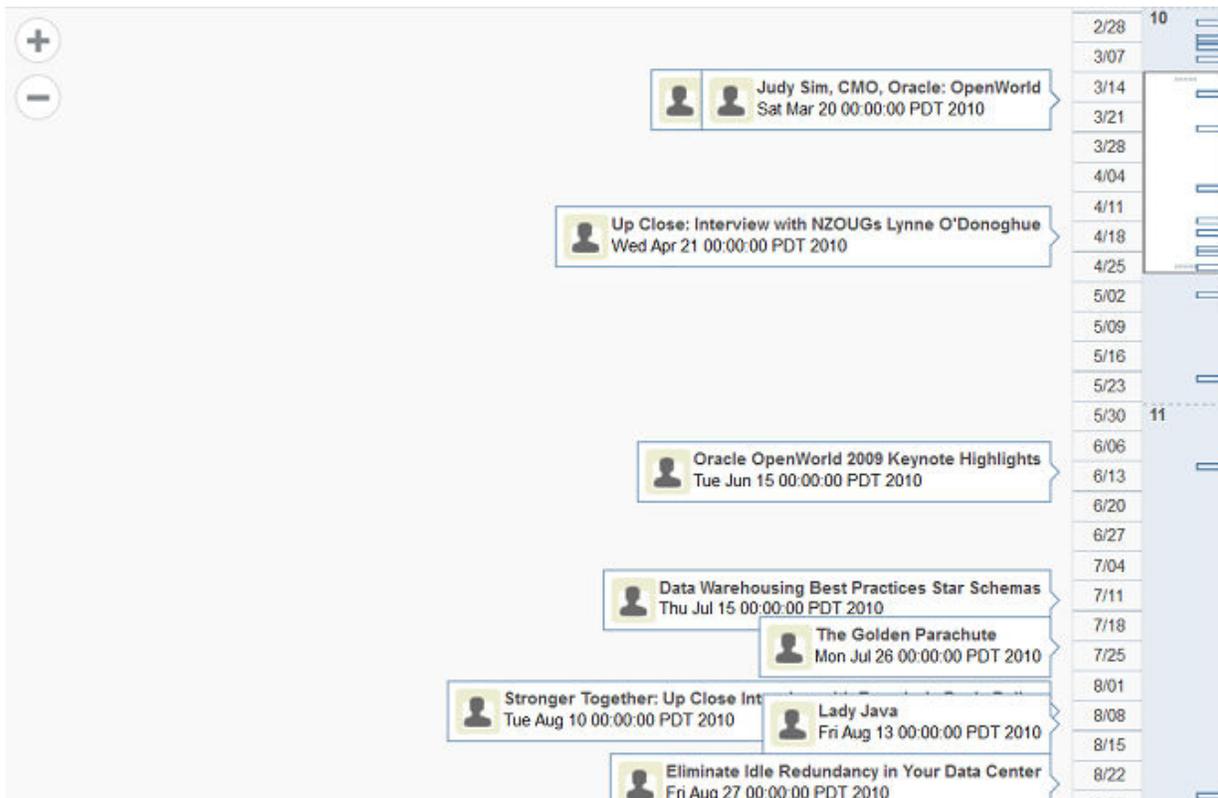


Layout Options

By default, timelines are displayed in a horizontal orientation with events laid out along a horizontal time axis and overview panel. You can change the layout to a vertical orientation with events displayed along a vertical time axis and overview panel. While you can specify that timeline items in a horizontal orientation will not overlap each other in the display, you cannot apply that configuration to items in a vertical orientation.

Figure 29-4 illustrates a timeline of speaking events, with a vertical orientation.

Figure 29-4 Timeline Horizontal and Vertical Orientations



Timeline Item Selection

Each event displayed in the timeline is represented as a timeline item that can include data display components such as images, text, and text labels, or actions such as links, buttons, and menus. A line feeler connects the event to the date in the time axis of the timeline. Events are represented in the overview panel as a configurable marker.

By default timeline items are not configured for selection at runtime. You can configure selection of a single or multiple timeline items. At runtime the event, the line feeler, and the marker in the overview panel are highlighted.

Content Delivery

Timelines can be configured for how data is delivered from the data source. The data can be delivered to the timeline either immediately upon rendering, as soon as the data is available, or lazily fetch after the shell of the component has been rendered. By default, timelines support the delivery of content from the data source when it is available. The `contentDelivery` attribute is set to `whenAvailable` by default.

Timelines are virtualized, meaning not all data on the server is delivered to and displayed on the client. You can configure timelines to fetch a certain number of rows or columns at a time from your data source based on date related values. Use `fetchStartTime` and `fetchEndTime` to configure fetch size.

Timeline Image Formats

Timelines support the following image formats: HTML5, Flash, and Portable Network Graphics (PNG). All image formats support locales using right-to-left display.

By default, timelines will display in the best output format supported by the client browser. If the best output format is not available on the client, the application will default to an available format. For example, if the client does not support HTML5, the application will use:

- Flash, if the Flash Player is available

You can control the use of Flash content across the entire application by setting a `flash-player-usage` context parameter in `adf-config.xml`. For more information, see [Configuring Flash as Component Output Format](#).

- PNG output format

Although static rendering, such as maintaining pan and zoom state of the Flash display, is fully supported when using the printable PNG output format, certain interactive features are not available including:

- Animation
- Context menus
- Drag and drop gestures
- Popup support
- Selection

Timeline Display in Printable or E-mailable Pages

ADF Faces allows you to output your JSF page from an ADF Faces web application in a simplified mode for printing or emailing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scrollbars and buttons. If a page is to be emailed, the page must be simplified so that email clients can correctly display it.

ADF Faces allows you to output your JSF page from an ADF Faces web application in a simplified mode for printing or emailing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scrollbars and buttons. If a page is to be emailed, the page must be simplified so that email clients can correctly display it. For information about creating simplified pages for these outputs, see [Using Different Output Modes](#).

When a timeline is displayed on a JSF page to be output in printable or e-mailable pages:

- Only the events currently in view on the timeline will be included in the content.
- In email mode, the events will be displayed as a table.
- In print mode, the timeline overview is not rendered.

Additional Functionality for Timeline Components

You may find it helpful to understand other ADF Faces features before you implement your `timeline` component. Additionally, once you have added a timeline to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that `timeline` components can use:

- **Partial page rendering:** You may want a timeline to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- **Personalization:** When enabled, users can change the way the timeline displays at runtime. Those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- **Accessibility:** Timeline components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- **Touch devices:** When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- **Content Delivery:** You can configure your timeline to fetch data from the data source immediately upon rendering the components, or on a second request after the components have been rendered using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- **Automatic data binding:** If your application uses the Fusion technology stack, then you can create automatically bound timelines based on how your ADF Business Components are configured. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how content is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using Timeline Components

To use the ADF DVT Timeline component, add the timeline to a page using the Component Palette window. Then define the data for the timeline and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Timeline Component Data Requirements

The data layer for the `timeline` component is specified in its child, the `timeSeries` component. You must specify at least one time series, at most two in the case of a dual timeline, using a model to access data from the underlying source.

The specific model class to use is an instance of `org.apache.myfaces.trinidad.model.CollectionModel`. This class extends the JSF `DataModel` class and adds on support for row keys. In the `DataModel` class, rows are identified entirely by index. However, to avoid issues if the underlying data changes, the `CollectionModel` class is based on row keys instead of indexes.

You may use other model instances, such as `java.util.List`, `java.util.ArrayList`, and `javax.faces.model.DataModel`. The timeline series component will automatically convert the instance into a `CollectionModel`, but without any additional functionality. For information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html.

Timelines require that the following attributes be set for the `timeSeries` component in JDeveloper:

- `value`: An EL Expression that references the data model represented in the timeline.
- `var`: The name of a variable to be used during the rendering phase to reference each element in the timeline collection. This variable is removed or reverted back to its initial value once rendering is complete.

Each immediate child of a `timeSeries` component must be at most one `timeItem` component. This component makes it possible to customize the event content through its `title`, `description` and `thumbnail` attributes.

Each time a time item is created, the value for the current item is copied into a `var` property, and optionally, additional data for the item is copied into a `varStatus` property. These properties can be accessed in EL expressions inside the time item component, for example, to pass the item value to the `title` attribute. Once the timeline has completed rendering, the `var` and `varStatus` properties are removed, or reverted back to their previous values.

The values for the `value`, `var`, and optionally, `varStatus` attributes must be stored in the timeline's data model or in classes and managed beans if you are using UI-first development.

The example below shows a code sample that adds a `TimelineCBBean` managed bean to your application that references the class or bean that contains the data, and

optionally, adds any other methods to customize the timeline. Not all list items in the data set specified by the `ArrayList` class are included in the example.

```
//imports needed by methods
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.Set;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.bean.RequestScoped;
import javax.faces.component.behavior.ClientBehavior;
import javax.faces.component.behavior.ClientBehaviorHint;
import javax.faces.event.AjaxBehaviorEvent;
import oracle.adf.view.faces.bi.component.timeline.UITimeSeries;
import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.ModelUtils;
import org.apache.myfaces.trinidad.model.RowKeySet;
@ManagedBean(name="cb")
public class TimelineCBBean
{
    private CollectionModel m_model;
    public TimelineCBBean()
    {
        super();
    }
    public CollectionModel getModel()
    {
        if (m_model != null)
            return m_model;
        ArrayList _list = new ArrayList(10);
        _list.add(new EmpEvent("0", parseDate("01.13.2010"), "Oracle
Application
        Express", "sel98AyXcsk", null));
        _list.add(new EmpEvent("1", parseDate("01.27.2010"), "Larry Ellison
on the
        Sun-Oracle Close", "ylNgcD2Ay6M", null));
        ...

        m_model = ModelUtils.toCollectionModel(_list);
        return m_model;
    }
    public void handleKey(AjaxBehaviorEvent event)
    {
        ClientBehavior _behavior = (ClientBehavior)event.getBehavior();
        Set<ClientBehaviorHint> _hints = _behavior.getHints();
        UITimeSeries _series =
            (UITimeSeries)event.getComponent().findComponent("ts1");
        if (_series == null)
            return;
        RowKeySet _rowKeySet = _series.getSelectedRowKeys();
        Iterator _iterator = _rowKeySet.iterator();
```

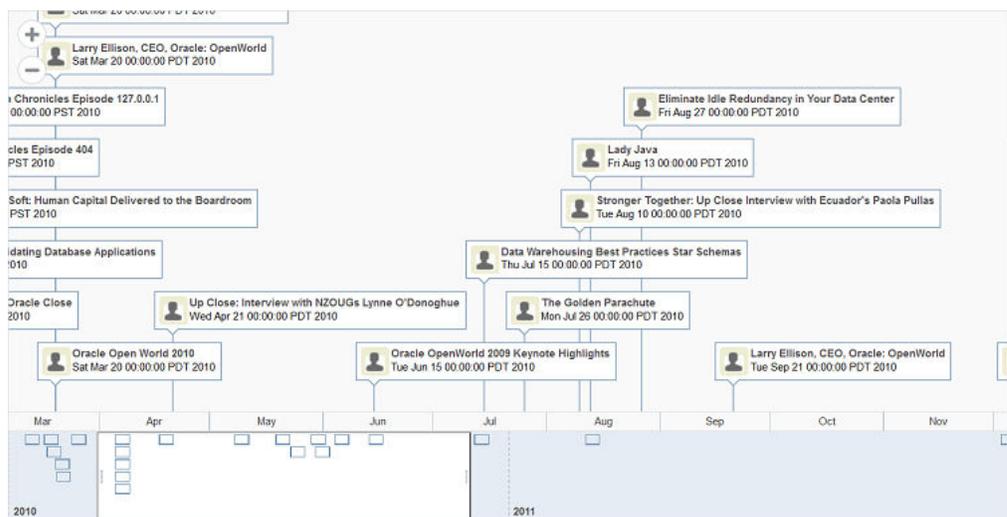
```

ArrayList _list = (ArrayList)m_model.getWrappedData();
while (_iterator.hasNext())
{
    Object _rowKey = _iterator.next();
    Object _event = m_model.getRowData(_rowKey);
    _list.remove(_event);
}
}
private static Date parseDate(String date)
{
    Date ret = null;
    try
    {
        ret = s_format.parse(date);
    }
    catch (ParseException e)
    {
        e.printStackTrace();
    }
    return ret;
}
static DateFormat s_format = new SimpleDateFormat("MM.dd.yyyy");

```

The managed bean example provides the data model for the Employee Presentations timeline displayed in [Figure 29-5](#). You can find the complete source code for the `TimelineCBBBean` in the ADF Faces Components Demo application. For information about the demo application, see [ADF Faces Components Demo Application](#).

Figure 29-5 Timeline of Employee Presentations



Configuring Timelines

The timeline component has configurable attributes and child components that you can add or modify to customize the display or behavior of the timeline. The prefix `dvt:`

occurs at the beginning of each timeline component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure timeline child components, attributes, and supported facets in the following areas:

- Timeline component (`timeline`): The parent component that wraps the timeline child components and facets.
- Timeline series (`timeSeries`): The immediate child of the timeline component used to specify the data layer for the timeline. You must specify at least one series in a timeline. You can also specify up to one additional series to be used for a comparison between timelines.

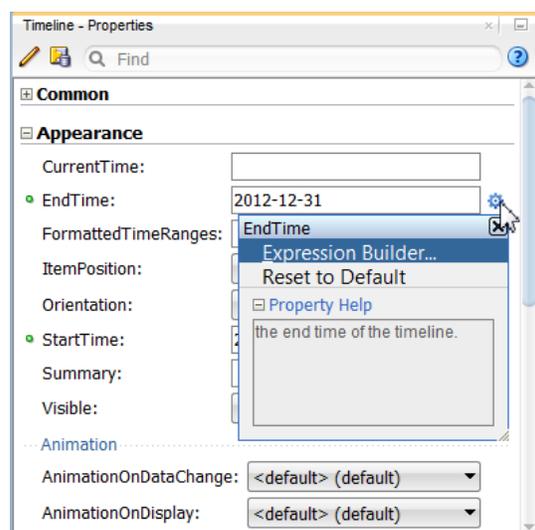
The timeline series component supports facets that can be used to configure context menus including:

- `bodyContextMenu`: Specifies a context menu that is displayed on non-selectable elements in the timeline component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the timeline component.
- Timeline item (`timeItem`): The child of `timeSeries` that represents an event in the timeline.
- Marker (`marker`): A configurable shape that represents the event in the overview panel. The attributes are specified in a named `overviewItem` facet child of the `timeItem` component.
- Time axis (`timeAxis`): Child of `timeline` used to specify the time axis and `timelineOverview` used to specify the overview time axis.
- Timeline overview (`timelineOverview`): An optional component used to provide a macro view of all of the events from all timeline series in the timeline. Users can scroll through the timeline using a zoom control.

How to Add a Timeline to a Page

When you are designing your page using UI-first development, you use the Components window to add a timeline to a JSF page. When you drag and drop a timeline component onto the page, a timeline artifact and source code is added to the Visual Editor, and the tag structure is added to the Structure window.

After the timeline is added to your page, you can use the Properties window to specify data values and configure display attributes. In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. [Figure 29-6](#) shows the dropdown menu for a timeline `endTime` attribute.

Figure 29-6 Timeline endTime Attribute Value

 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a timeline and the binding will be done for you. For more information, see the "Creating Databound Timelines" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how timeline attributes and timeline child tags can affect functionality. For more information, see [Configuring Timelines](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Timeline Components](#).

To add a timeline to a Page:

1. In the ADF Data Visualization page of the Components window, from the Gantt section, drag and drop a `Timeline` component onto the page.
2. In the Properties window, view the attributes for the timeline. Use the help button to display the complete tag documentation for the `timeline` component.
3. Expand the Appearance section, and enter values for the following attributes:
 - **EndTime:** Enter the ending date to use for the timeline time range using the format `yyyy-mm-dd`. Select an end date that will include events in the data collection you wish to display on the timeline. By default the current date is used for this attribute.
 - **StartTime:** Enter the starting date to use for the timeline time range using the format `yyyy-mm-dd`. Select a start date that will include events in the data collection you wish to display on the timeline. By default the current date is used for this attribute.

4. Optionally, enter values for the following attributes:
 - **Orientation:** Use the attribute's dropdown menu to change the default layout from `horizontal` to `vertical`.
For sample images of timeline orientation, see [Layout Options](#).
 - **ItemPosition:** If you are using a vertical orientation for the timeline, by default timeline items will not overlap each other vertically in the available space for the timeline. The default value is `noOverlap`. In a vertical orientation, this attribute does not apply to the horizontal display of timeline items.
You can use an attribute value of `random` to specify that timeline items will randomly lay out the items vertically in the available space for the timeline.
 - **Summary:** Enter a summary of the timeline's purpose and structure for screen reader support.
 - **TimeZone:** Enter the time zone to use for the timeline. If not set, the value is identified from the `AdfFacesContext`.
5. Expand the Behavior section, and optionally enter values for the following attributes:
 - **ItemSelection:** Use the dropdown list to specify whether or not timeline items in the timeline are selectable. Valid values are `single` (default), `multiple`, or `none`. This setting applies to both timeline series in a dual timeline.
 - **SortData:** Use to set whether timeline events are sorted automatically by the timeline based on the time of the event, or manually sorted by the data model to which it is bound. Valid values are `auto` (default) or `none`.
 - **FetchStartTime** and **FetchEndTime:** Use these attributes to specify the start and end dates to use for delivering content from the data source.
6. To set the time axis for the timeline, do the following:
 - a. In the Structure window, right-click the `timeline` node and select **Insert Inside Timeline > Time Axis**.
 - b. In the Insert Time Axis dialog, enter the scale to use for the time axis of the timeline. Valid values are `twoyears`, `years`, `quarters`, `twomonths`, `months`, `twoweeks`, `weeks`, `days`, `sixhours`, `threehours`, `hours`, `halfhours`, and `quarterhours`.
7. To add an optional timeline time axis to the timeline, do the following:
 - a. In the Structure window, right-click the `timeline` node and select **Insert Inside Timeline > Timeline Overview**.
 - b. In the Structure window, right-click the `timelineOverview` node and select **Insert Inside TimelineOverview > Time Axis**.
 - c. In the Insert Time Axis dialog, enter the scale to use for the overview time axis display of the timeline. Valid values are `twoyears`, `years`, `quarters`, `twomonths`, `months`, `twoweeks`, `weeks`, `days`, `sixhours`, `threehours`, `hours`, `halfhours`, and `quarterhours`.

What Happens When You Add a Timeline to a Page

JDeveloper generates a single timeline tag and a default axis tag when you drag and drop a timeline from the Components window onto a JSF page without setting any additional attributes in the Properties window.

The example below shows the generated code.

```
<dvt:timeline startTime="2017-02-21" endTime="2017-02-21" id="t1">  
  <dvt:timeAxis scale="months" id="ta5"/>  
</dvt:timeline>
```

If you choose to use the Data Controls panel to bind the data to a data control when creating the timeline, JDeveloper generates code based on the data model. See [Creating Databound Gantt Chart and Timeline Components in *Developing Fusion Web Applications with Oracle Application Development Framework*](#).

Adding Data to Timeline Components

You can add data to the ADF DVT Timeline using UI-first development by creating classes and managed beans, and then using methods to automate the creation of the tree model and reference the data classes and beans. Timeline components require a collection data model to display attributes.

For example, to create the Employee Presentation timeline illustrated in [Figure 29-5](#), you must provide a data model that includes a qualifying date value and details about the events.

How to Add Data to a Timeline

To add data to the timeline using UI-first development, create the classes, managed beans, and methods that will create the model and reference the classes, beans, or methods in JDeveloper.

Before you begin:

It may be helpful to have an understanding of how timeline attributes and timeline child tags can affect functionality, as described in [Configuring Timelines](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features, as described in [Additional Functionality for Timeline Components](#).

Add a timeline to your page. For help, see [How to Add a Timeline to a Page](#). Confirm that the start time and end time for the timeline is consistent with the data model you are using.

Create the classes and managed beans that will define the timeline's data model and supply the data to the timeline. For information and examples, see [Timeline Component Data Requirements](#). If you need help creating classes, see [Working with Java Code in *Developing Applications with Oracle JDeveloper*](#). For help with managed beans, see [Creating and Using Managed Beans](#).

To add data to the timeline in UI-first development:

1. In the Structure window, right-click the `timeline` node and choose **Insert Inside Timeline > Time Series**.
2. Right-click the `timeSeries` node and choose **Go to Properties**.
3. In the Properties window, expand the Common section, and set the following attributes:

- **Value:** Specify an EL expression for the model to which you want the timeline to be bound. This must be an instance of `org.apache.myfaces.trinidad.model.CollectionModel`.

 **Note:**

You may use other model instances, such as `java.util.List`, `array`, and `javax.faces.model.DataModel`. The timeline component will automatically convert the instance into a `CollectionModel`.

For example, reference the managed bean you created to instantiate the timeline. In the employee presentation example, the timeline managed bean is named `cb`, and the data is instantiated when the timeline is referenced. To use the employee presentation data example with a timeline, enter the following in the **Value** field for the EL expression: `#{cb.Model}`.

- **Var:** Enter the name of a variable to be used during the rendering phase to reference each element in the timeline collection. This variable is removed or reverted back to its initial value once rendering is complete.

For example, enter `evt` in the **Var** field to reference each element in the employees presentation data example.

- **VarStatus:** Optionally, enter the name of a variable during the rendering phase to access contextual information about the state of the component, such as the collection model or loop counter information. This variable is removed or reverted back to its initial value once rendering is complete.
4. In the Structure window, right-click the `timeSeries` node and choose **Insert Inside Time Series > Time Item** to add a component to display the timeline series data.
 5. In the Properties window for the `dvt:timeItem`, expand the Common section and enter the following values

- **Value:** Enter an EL Expression that references the date-related value you wish to display as an item on the timeline. For example, in a collection of date-related employee presentations, you could display presentation date as a timeline item.

For example, to reference the employee presentations data source, enter `#{evt.date}`.

- **EndTime:** Optionally, you can configure timeline items to represent a duration in time. By adding an end time to the value of the timeline item, you can represent a span of time instead of a single instant in time.
- **Title:** Enter an EL Expression that references the main text output for each date-related value in the time item. For example, in a collection of date-related employee presentations, you could display the presentation subject as a title.
To use the presentation subject as suggested, enter `#{evt.description}`.
- **Description:** Enter an EL Expression that references the subtext output for each date-related value in the time item. For example, in a collection of date-related employee presentations, you could display the presentation date and time as a description.

To use the date for the subtitle as suggested, enter `#{evt.date}`.

- **Thumbnail:** Enter the image URL to use as a thumbnail for every individual time item. If you have to use multiple images depending on your time items, you can use an EL expression in the URL.

For example, to use the categories of presentations for the thumbnail, enter `/resources/images/timeline/#{evt.type}.png"/`, where the specified images folder contains files that match the names of the presentation types in the data collection. The EL expression will evaluate to the type name and search for a PNG file of the same name to use as the item thumbnail.

6. To configure the marker representing the timeline item that displays in the timeline overview panel, do the following:
 - a. In the Structure window, right-click the `overviewItem` facet and select **Insert Inside f:facet-overviewItem > Marker**.
 - b. In the Properties window, set values to specify shape, size, and fill color as desired. By default, the marker displayed in the overview panel is a square shape.

For help with creating EL expressions, see [How to Create an EL Expression](#).

For information about timeline items, see [Configuring a Timeline Item Duration](#).

What You May Need to Know About Configuring Data for a Dual Timeline

You can add up to one additional timeline series to configure a dual timeline to compare two series of events. The procedure for adding and configuring another `timeSeries` component is the same.

The following example shows the code for the dual timeline displayed in [Figure 29-2](#):

```
<af:panelGroupLayout layout="horizontal">
  <af:selectOneChoice id="soc" label="Employee (TOP)"
value="#{timeline.firstEmp}"
      autoSubmit="true"
      valueChangeListener="#{timeline.handleValueChange}">
    <af:forEach items="#{timeline.employees}" var="ce">
      <af:selectItem label="#{ce.name}" value="#{ce.id}"/>
    </af:forEach>
  </af:selectOneChoice>
  <af:spacer width="5"/>
  <af:outputText value="vs."/>
  <af:spacer width="5"/>
  <af:selectOneChoice id="soc2" label="Employee (BOTTOM)"
value="#{timeline.secondEmp}"
      autoSubmit="true"
      valueChangeListener="#{timeline.handleValueChange}">
    <af:forEach items="#{timeline.employees}" var="ce">
      <af:selectItem label="#{ce.name}" value="#{ce.id}"/>
    </af:forEach>
  </af:selectOneChoice>
</af:panelGroupLayout>

<dvt:timeline id="tl1" startTime="2000-01-01" endTime="2011-12-31"
inlineStyle="width:1024px;height:500px">
```

```

itemSelection="single"
    currentTime="2010-04-01">
  <dvt:timeSeries id="ts1" var="evt" value="{timeline.firstModel}"
    contentDelivery="lazy">
    <dvt:timeItem id="ti1" value="{evt.date}" group="{evt.group}"
      title="{evt.description}" description="{evt.date}"
      thumbnail="/resources/images/timeline/{evt.type}.png">
      <f:facet name="overviewItem">
        <dvt:marker id="m1" shape="circle" fillColor="#ff0000"/>
      </f:facet>
    </dvt:timeItem>
  </dvt:timeSeries>
  <dvt:timeSeries id="ts2" var="evt" value="{timeline.secondModel}"
    contentDelivery="lazy">
    <dvt:timeItem id="ti2" value="{evt.date}"
      title="{evt.description}" description="{evt.date}"
      thumbnail="/resources/images/timeline/{evt.type}.png">
      <f:facet name="overviewItem">
        <dvt:marker id="m2" shape="circle" fillColor="#0000ff"/>
      </f:facet>
    </dvt:timeItem>
  </dvt:timeSeries>
  <dvt:timeAxis id="ta1" scale="quarters"/>
  <dvt:timelineOverview id="ov1">
    <dvt:timeAxis id="ta2" scale="years"/>
  </dvt:timelineOverview>
</dvt:timeline>

```

What You May Need to Know About Adding Data to Timelines

The examples in this chapter use classes and managed beans to provide the data to the timeline. If your application uses the Fusion technology stack, then you can use data controls to create a timeline and the binding will be done for you.

For more information, see the "Creating Databound Gantt Charts and Timelines" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Alternatively, if you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Customizing Timeline Display Elements

You can configure ADF DVT Timeline items using a number of customization options and attributes. You can also add a custom time scale to your timeline and its overview window.

Customizing Timeline Items

Time items represent the events displayed in the timeline. The `timeItem` component does not currently support customization via stamping.

Time items are represented in the timeline overview panel as a configurable shape. You can specify the following attributes for a time item marker:

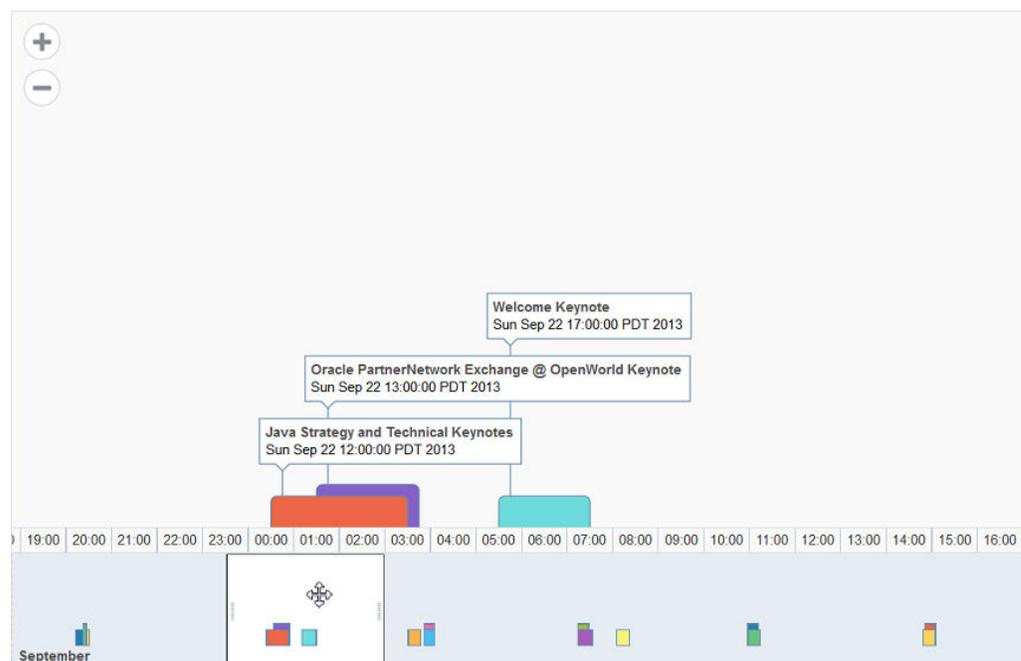
- `fillColor`: The color of the marker shape. Valid values are RGB hexadecimal colors.
- `opacity`: The opacity of the fill color of the marker. Valid values range from 0 percent for transparent, to 100 percent for opaque.
- `shape`: The shape of the overview marker for each selected timeline series value. Valid values are one of seven prebuilt shapes `circle` (default), `diamond`, `human`, `plus`, `square`, `triangleDown`, and `triangleUp`. This attribute is not supported for timelines with a vertical orientation.
- `scaleX` and `scaleY`: The `scaleX` (horizontal) and `scaleY` (vertical) scale factor. Valid value is a numerical percentage. JDeveloper will automatically resize a marker to fit within the timeline overview area if the marker is too large. These attributes are not supported for timelines with a vertical orientation.

Configuring a Timeline Item Duration

Instead of specifying a specific instance of time for a timeline item you can configure a span or duration of time.

For example, [Figure 29-7](#) shows a timeline with timeline items that span one or more hours in duration and display as a bar. The timeline overview panel also represents the timeline items using a bar display.

Figure 29-7 Timeline Items with Time Durations



To specify a timeline duration bar, set both the `value` and `endTime` attributes for the `timeItem` component. The values must be stored in the timeline's data model or in classes and managed beans if you are using UI-first development. In the following sample code, the timeline item dates are configured as a duration bar:

```
<dvt:timeline id="tl1" startTime="2013-09-21" endTime="2013-09-27"
itemSelection="multiple"
                                orientation="horizontal"
summary="Timeline Durations Demo">
  <dvt:timeSeries id="ts1" var="evt" value="{timeline.durationModel}">
    <dvt:timeItem id="ti1" value="{evt.date}" endTime="{evt.endDate}"
                                title="{evt.description}"
description="{evt.date}"/>
  </dvt:timeSeries>
  <dvt:timeAxis id="ta1" scale="hours" zoomOrder="weeks days hours"/>
  <dvt:timelineOverview id="ov1">
    <dvt:timeAxis id="ta2" scale="months"/>
  </dvt:timelineOverview>
</dvt:timeline>
```

How to Add a Custom Time Scale to a Timeline

You can create a custom time scale for the timeline and overview axes. The custom time scale is configured in the `scale` attribute of the `dvt:timeAxis`.

Before you begin:

It may be helpful to have an understanding of how timeline attributes and timeline child components can affect functionality. For more information, see [Configuring Timelines](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Timeline Components](#).

You should already have a timeline on your page. If you do not, follow the instructions in this chapter to create a timeline. For information, see [How to Add a Timeline to a Page](#).

To create and use a custom time axis:

1. Implement the `CustomTimescale.java` interface to call the method `getNextDate(Date currentDate)` in a loop to build the time axis. The example below shows sample code for the interface.

```
public interface CustomTimescale
{
    public String getScaleName();
    public Date getPreviousDate(Date timelineStartDate);
    public Date getNextDate(Date currentDate);
    public String getLabel(Date date);
}
```

2. In the Structure window, right-click a timeline node and choose **Go to Properties**.

3. Expand the **Advanced** category of the Properties window, for the **CustomTimeScales** attribute, register the implementation of the interface for the custom time axis.

The `customTimeScales` attribute's value is a `java.util.Map` object. The specified map object contains pairs of key/values. The key is the time scale name (`fiveyears`), and the value is the implementation of the `CustomTimeScale.java` interface. For example:

```
customTimeScales="#{timeline.customTimescales}"
```

4. To use the custom time scale in the time axis or overview time axis, in the Structure window, right-click the `dvt:timeAxis` node and in the Properties window, enter the custom time scale name.

The example below shows sample code for setting a `threeyears` time axis and a `fiveyears` overview time axis.

```
<dvt:timeline>
  <dvt:timeAxis id="ta1" scale="threeyears"/>
  <dvt:timelineOverview id="ov1">
    <dvt:timeAxis id="ta2" scale="fiveyears"/>
  </dvt:timelineOverview>
</dvt:timeline>
```

What You May Need to Know About Skinning and Customizing the Appearance of Timelines

For the complete list of timeline skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactive Features to Timelines

You can add interactive features to ADF DVT Timeline components, including support for popups, custom context menus, and drag and drop operations.

How to Add Popups to Timeline Items

Timeline `timeItem` components can be configured to display popup dialogs, windows, and menus that provide information or request input from end users. Using the `af:popup` component with other ADF Faces components, you can configure functionality to allow your end users to show and hide information in secondary windows, input additional data, or invoke functionality such as a context menu.

With ADF Faces components, JavaScript is not needed to show or hide popups. The `af:showPopupBehavior` tag provides a declarative solution, so that you do not have to write JavaScript to open a `popup` component or register a script with the `popup` component. For more information about these components, see [Using Popup Dialogs, Menus, and Windows](#).

Configuring Timeline Context Menu

You can define timeline context menus using these context menu facets:

- `bodyContextMenu`: Specifies a context menu that is displayed on non-selectable elements in the timeline component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the timeline component.

Each facet supports a single child component. Context menus are currently only supported in Flash and HTML5.

You create a context menu by using `af:menu` components within an `af:popup` component. You can then invoke the context menu popup from another component, based on a specified trigger. For more information about configuring context menus, see [Using Popup Dialogs, Menus, and Windows](#).

Due to technical limitations when using the Flash rendering format, context menu contents are currently displayed using the Flash Player's context menu. This imposes several limitations defined by the Flash Player:

- Flash does not allow for submenus in its context menu.
- Flash limits custom menu items to 15. Any built-in menu items for the component, for example, a pie graph `interactiveSliceBehavior` menu item, will count towards the limit.
- Flash limits menu items to text-only. Icons or other controls possible in ADF Faces menus are not possible in Flash menus.
- Each menu caption must contain at least one visible character. Control characters, new lines, and other white space characters are ignored. No caption can be more than 100 characters long.
- Menu captions that are identical to another custom item are ignored, whether the matching item is visible or not. Menu captions are compared to built-in captions or existing custom captions without regard to case, punctuation, or white space.
- The following captions are not allowed, although the words may be used in conjunction with other words to form a custom caption: **Save, Zoom In, Zoom Out, 100%, Show All, Quality, Play, Loop, Rewind, Forward, Back, Movie not loaded, About, Print, Show Redraw Regions, Debugger, Undo, Cut, Copy, Paste, Delete, Select All, Open, Open in new window, and Copy link.**
- None of the following words can appear in a custom caption on their own or in conjunction with other words: **Adobe, Macromedia, Flash Player, or Settings.**

Additionally, since the request from Flash for context menu items is a synchronous call, a server request to evaluate EL is not possible when the context menu is invoked. To provide context menus that vary by selected object, the menus will be pre-fetched if the context menu popup uses the setting `contentDelivery="lazyUncached"`. For context menus that may vary by state, this means that any EL expressions within the menu definition will be called repeatedly at render time, with different selection and currency states. When using these context menus that are pre-fetched, the application must be aware of the following:

- Long running or slow code should not be executed in any EL expression that may be used to determine how the context menu is displayed. This does not apply to

`af:commandMenuItem` attributes that are called after a menu item is selected, such as `actionListener`.

- In the future, if the Flash limitations are solved, the ADF context menu may be displayed in place of the Flash context menu. To ensure upgrade compatibility, you should code such that an EL expression will function both in cases where the menu is pre-fetched, and also where the EL expression is evaluated when the menu is invoked. The only component state that applications should rely on are `selection` and `currency`.

Using Map Components

This chapter describes how use ADF Faces Data Visualization `map` and `thematicMap` components to display data in geographic and thematic maps using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the Fusion technology stack, then you can also use data controls to create geographic map themes and thematic map area and point data layers to display data. For more information, see the "Creating Databound Geographic and Thematic Map Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About Map Components](#)
- [Using the Geographic Map Component](#)
- [Customizing Geographic Map Display Attributes](#)
- [Customizing Geographic Map Themes](#)
- [Adding a Toolbar to a Geographic Map](#)
- [Using Thematic Map Components](#)
- [Defining Thematic Map Base Maps](#)
- [Customizing Thematic Map Display Attributes](#)
- [Adding Interactive Features to Thematic Maps](#)

About Map Components

An ADF DVT Geographic Map represents business data in one or more interactive layers of information (known as themes), superimposed on a single map. An ADF DVT Thematic Map represents business data as patterns in stylized areas or associated markers.

Geographic maps require a configuration that contains a URL to a remote Oracle Application Server (AS) MapViewer service, and optionally, a geocoder service if address data will have to be converted to longitude and latitude.

Thematic maps do not require a connection to an Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map.

Map Component Use Cases and Examples

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed.

Both geographic and thematic maps are designed to display data. Geographic maps focus on data that is best displayed with details such as roads or rivers, and requires configuration to an Oracle MapViewer service and optionally, a geocoder service. Thematic maps focus on data trends or patterns without the visual clutter of geographic details and do not require configuration to an Oracle MapViewer or geocoder service.

Geographic maps support a variety of map themes, each of which must be bound to a data collection. The following kinds of map themes are available:

- **Color:** Applies to regions. For example, a color theme might identify a range of colors to represent the population in the states of a region or the popularity of a product in the states of a region. A geographic map can have multiple color themes visible at different zoom levels. For example, a color theme at zoom levels 1 to 5 might represent the population of a state, and the county median income at zoom levels 6 to 10.
- **Point:** Displays individual latitude/longitude locations in a map. For example, a point theme might identify the locations of warehouses in a map. If you customize the style of the point that is displayed, you might choose to use a different image for the type of inventory (electronics, housewares, garden supplies) in a set of warehouses to differentiate them from each other.
- **Graph:** Creates any number of pie graph themes and bar graph themes. However, only one graph theme can be visible at a given time. You select the desired theme from the **View** menu of the map toolbar. Graph themes can show statistics related to a given region such as states or counties. For example, a graph theme could display the sales values for a number of products in a state.

[Figure 30-1](#) displays a geographic map with color, point, and graph themes.

Figure 30-1 Geographic Map of Southwest US with Color, Point, and Pie Graph Themes

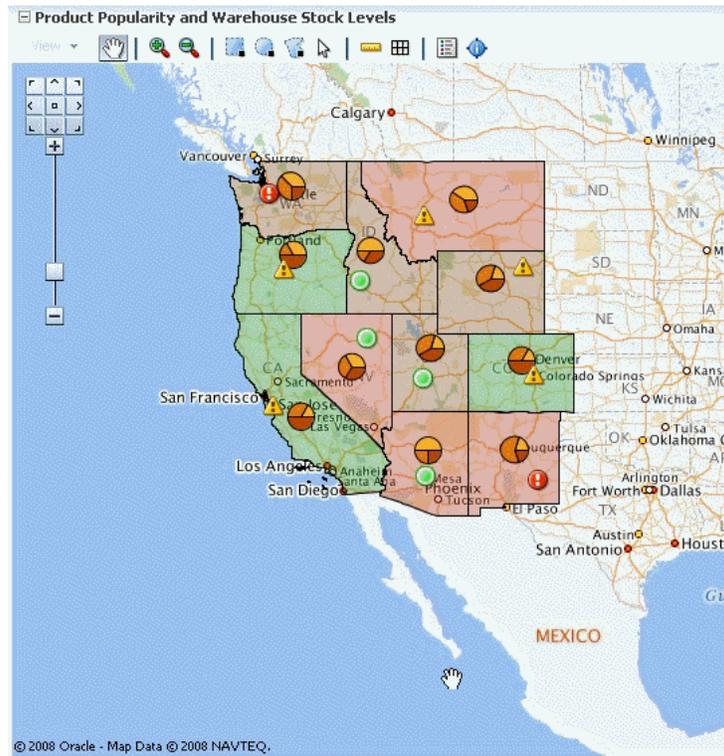
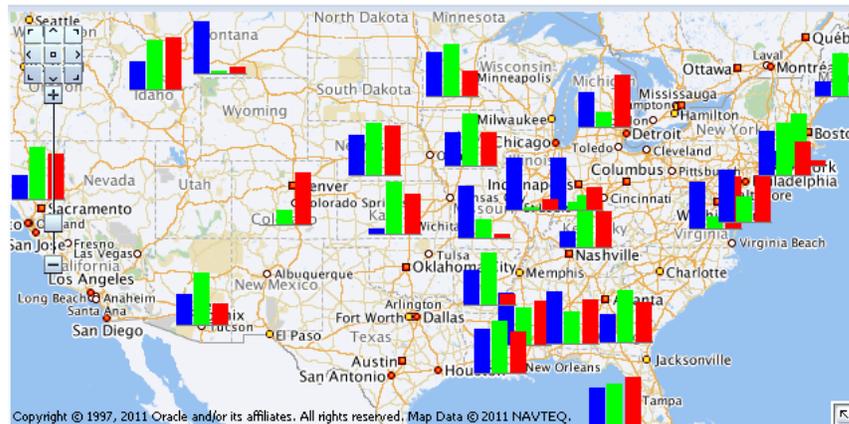


Figure 30-2 shows a geographic map with a customized bar graph theme.

Figure 30-2 Geographic Map with Custom Bar Graph Theme



Thematic maps display trends or patterns in data associated with a geographic location. Data is stylized by region, for example, using a fill color based on data values, associating a marker with the region, or both.

Figure 30-3 shows a thematic map that displays unemployment rates by states in the US. The map displays multiple selection of the states with an employment rate of 2.0-4.0 percent.

Figure 30-3 Thematic Map of Unemployment Rates in the US

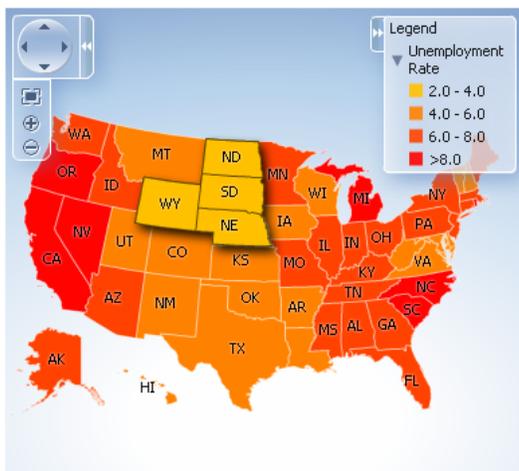


Figure 30-4 shows a thematic map that displays a graduated symbol for the size of the major cities in South America.

Figure 30-4 Thematic Map with City by Size in South American



End User and Presentation Features of Maps

The ADF Data Visualization map and thematic map components provides a range of features for end users, such as panning and zooming and legend display. It also provides a range of presentation features, such as state management.

Geographic Map End User and Presentation Features

To understand how geographic maps are used and can be customized, it may be helpful to review these elements and features:

- **Viewport:** The container for the geographic map and its presentation features. The default size is 600px by 375px and is customizable.
- **Base map:** The background geographic data, zoom levels, and the appearance and presence of items such as countries, cities, and roads. By default, geographic maps use based maps from the remote Oracle MapViewer service. The base map can be any image that can be configured using a map viewer and map builder, for example, the floor maps of office buildings.
- **Zoom control:** Pan icons and a zoom slider that render in the upper left-hand corner of the map. [Figure 30-5](#) shows a map zoom control that is zoomed out all the way (that is, the zoom level is set to 0). At zero, the entire map is displayed.

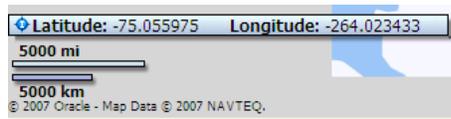
You can customize the location and the initial setting of the zoom control in the `map` component. The **View** menu in the map toolbar lets you determine the visibility of the zoom control. By default, the initial zoom level for a map is set to 0.

Figure 30-5 Zoom Control of a Map



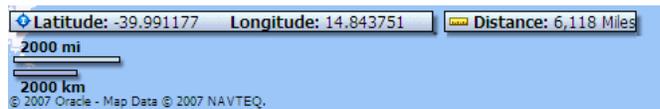
- **Pan icons:** Icons (with arrows that point north, south, east, west, northeast, northwest, southeast, and southwest) at the top of the zoom control. You can use these icons to move the entire map in specific directions.
- **Zoom slider:** Slider with a thumb for large scale zooming and icons for zooming a single level. You can use the plus icon to zoom in and the minus icon to zoom out one level at a time. When the thumb is at the bottom of the slider, the zoom level is zero.
- **Scale:** Two horizontal bars that display in the lower left-hand corner of the map below the information panel and above the copyright. [Figure 30-6](#) shows the scale. The top bar represents miles (mi) and the bottom bar represents kilometers (km). Labels appear above the miles bar and below the kilometers bar in the format: [distance] [unit of measure]. The length and distance values of the bars change as the zoom level changes and as the map is panned.

Figure 30-6 Map Information Panel, Scale, and Copyright



- Information panel: Displays latitude and longitude in the lower left-hand corner above the scale. [Figure 30-6](#) shows the information panel. By default, the information panel is not visible. You can display this panel from the **View** menu or by clicking the Information button on the toolbar.
- Measurement panel: Displays either distance, area, or radius depending on which tools in the toolbar are currently in use. Text appears in the following format: [label] [value] [unit of measure] to the right of the information panel. [Figure 30-7](#) shows the measurement panel with a distance measurement. Area measurement and radius measurement appear in a similar manner with the appropriate labels.

Figure 30-7 Map Measurement Panel Beside the Information Panel



The following tools provide information in the measurement panel:

- Area measurement: Appears only when the Area, Rectangular Selection, or Multi-Point Selection tools are active.
- Distance measurement: Appears only when the Distance tool is active.
- Radius measurement: Appears only when the Circular Selection tool is active.
- Copyright: Appears in the lower left-hand corner of the map and contains text that you can customize in the `map` component.
- Overview map: Consists of a miniature view of the main map as shown in [Figure 30-8](#). This map appears in the lower right-hand corner of the main map and lets you change the viewable region of the main map without having to use the pan tool or the pan icons.

Figure 30-8 Overview Map



The following items are part of the overview map:

- Reticule: Appears in a small window that you can move across a miniature view of the main map. The position of the reticule in the miniature map

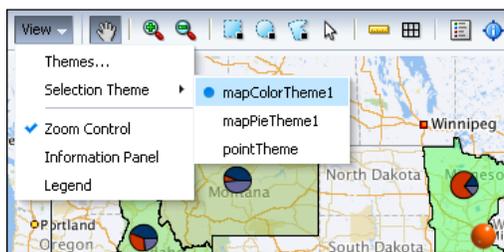
determines the viewable area of the main map. As you move the reticule, the main map is updated automatically.

- Show/Hide icon: Appears in the upper left-hand corner when the overview map is displayed. When you click the **Show/Hide icon**, the overview map becomes invisible and only the icon can be seen in the lower right corner of the main map.
- Toolbar: Appears in association with the map to provide user controls to adjust the display of the map and map themes.

The toolbar contains the following elements in the sequence listed:

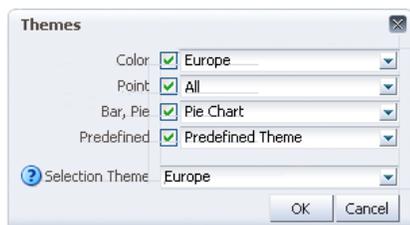
- View menu: Lets the user control which themes are visible, select a specific theme for display, and determine the visibility of the zoom control, information panel, and the legend. [Figure 30-9](#) show a sample **View** menu.

Figure 30-9 Map View Menu



The **Themes** option on the **View** menu provides a dialog to configure a geographic map when multiple themes are available. [Figure 30-10](#) shows a sample map themes dialog.

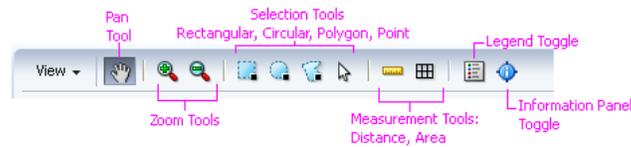
Figure 30-10 Map Themes Dialog



- Toolbar buttons: Provides user controls to interact with the geographic map including:
 - * Pan: Use to pan the map by drag and drop operation within the view
 - * Zoom: Use to zoom in or out in the map view
 - * Selection: Use to select shaped (rectangle, circle, or polygon) areas or points on the map. The themes within the selection will be highlighted
 - * Measurement: Use to hide or show the map distance and area measurements tools
 - * Legend: Use to hide or show the map legend

- * Information panel: Use to hide or show the map information panel
- Figure 30-11 shows the features supported by the map toolbar buttons

Figure 30-11 Map Toolbar Button Features



Thematic Map End User and Presentation Features

To understand how thematic maps are used and can be customized, it may be helpful to review these elements and features:

- Viewport: The container for the thematic map and its presentation features. The default size is 600px by 375px and is customizable.
- Base map: The background geographic area. Each base map includes several sets of regions and one fixed set of cities referred to as points. A set of regions or cities is referred to as a layer. Only one base map and one layer may be displayed at a time, with the exception of enabled drilling. The base maps prebuilt for the thematic map include:
 - United States
 - United States and Canada
 - World
 - Africa
 - Asia
 - Europe
 - North America
 - Latin America
 - South America
 - APAC (Asia Pacific)
 - EMEA (Europe, Middle East, and Africa)
 - World Regions

Note:

The thematic map component supports custom base maps. For more information, see [Defining Thematic Map Base Maps](#).

- Area layers and labels: Each base map includes several regions that represent levels in a geographical hierarchy. For example, in the United States map the hierarchy is country > states > counties. The hierarchical relationship supports drilling in the region. When displayed, the cities associated with the base map will

appear as data points. By default, each region in an area layer displays a label that can be customized.

Thematic maps support one or more custom area layers, a named group of regions based on either an existing area layer or another custom region. A custom region fits naturally into the geographical hierarchy of the base map. For example, a custom area layer based on the Counties layer would appear between the US States and US Counties layers in the hierarchy.

- Data layers: Each thematic map data layer is bound to a data collection and represents the data as an area, or a marker, or both. A map layer may display only one data layer at a time, whereas multiple data markers may appear at the same time. There are two types of data layers:
 - Area data layers: Area definitions associated with a geographic area, or by points associated with a map position as in longitude and latitude, or x and y coordinates. Data is stylized with color and pattern fills, or a data marker, or both.
 - Point data layers: Point locations associated with a map position as in longitude and latitude, or x and y coordinates. Data is represented by a marker or image using shapes available with the `thematicMap` component, custom shapes, or graphic files.
- Control panel: Optional tool that allows user to control the following operations:
 - Pan and zoom control: Use to pan the map, and zoom and center the map within the view.
 - Zoom to fit: Use to center and fit the full view of the map within the viewport.
 - Zoom buttons: Use to zoom in or out on the view of the thematic map.
 - Show/hide control panel button: Use to show or hide the control panel.
 - Reset button: Available when drilling is enabled for the thematic map. Use to reset map to display with no drilling in regions.
 - Drill buttons: Available when drilling is enabled for the thematic map. Use arrows pointing up or down to drill up or down in a map region.

Figure 30-12 shows the control panel with drilling enabled in the map.

Figure 30-12 Control Panel with Drilling Enabled

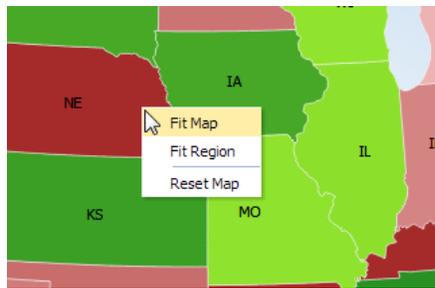


- Context menus: By default, thematic maps display context menus for the viewport background, map regions, and data markers. Custom context menu items can be added to the default menus.

Figure 30-13 shows the default context menu for the map viewport. The same menu items are available for data markers.

Figure 30-13 Map Viewport Background Context Menu

Figure 30-14 shows the context menu for a map region.

Figure 30-14 Map Region Context Menu

- State management: By default, display changes made to a thematic map such as center, zoom, selection, and drill state persist across sessions, and carry over to printing.
- Image formats: By default, thematic maps will display in the best output format supported by the client browser. If the best output format is not available on the client, the application will default to an available format. Thematic maps support the following image formats: HTML5, Flash, and Portable Network Graphics (PNG). All image formats support locales with right-to-left display.
- Printing: Thematic maps are printed using a PNG output format, maintaining any zoom or pan state in the map.
- Animation: By default, thematic maps are animated upon initial rendering of the map, when the data associated with the map changes, and when a region is drilled in the map.
- Drilling: When enabled, drilling of the next layer in the geographical hierarchy is displayed. For example, the counties within a US state are displayed when the user double-clicks a state region. Drilling icons are added to the Control Panel when drilling is enabled.
- Drag and drop: Maps can be configured to support these operations:
 - Drag selected map regions or data markers to another page component.
 - Move data markers from one location to another on the map.
 - Drag data elements from another page component to a map region or data marker.
- Disable features: End user features including map zoom, zoom-to-fit, and pan can be disabled for a thematic map. When disabled, the controls are removed from the control panel.

- **Tooltips:** By default, thematic maps use tooltips to orient the user to the map location and associated data when moving the cursor over the map.
- **Zooming:** Thematic maps can be configured to scale markers on zoom, to fit data on initial zoom, and to isolate data layer zooming.

Additional Functionality for Map Components

You may find it helpful to understand other ADF Faces features before you implement your map component. Additionally, once you have added a map component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that map components can use:

- **Partial page rendering:** You may want a map to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- **Personalization:** When enabled, users can change the way the map displays at runtime, and those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- **Accessibility:** Geographic and thematic map components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- **Skins and styles:** You can customize the appearance of gauge components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- **Content Delivery:** You can configure your thematic map area and point data layers to fetch a certain number of rows from your data source using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- **Automatic data binding:** If your application uses the Fusion technology stack, then you can create automatically bound maps based on how your ADF Business Components are configured. For more information, see the "Creating Databound Geographic and Thematic Map Components" chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

Note:

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and

how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Geographic Map Component

When you create an ADF DVT Geographic map, you are prompted to select a base map that an administrator has already configured using the Map Builder tool of Oracle Spatial. During configuration, the map administrator defines the zoom levels that the map supports. These levels also determine the zoom capability of the geographic map.

Administrators also have the option of creating predefined map themes using the Map Builder tool. For example, a predefined theme might use specific colors to identify regions. In the geographic map component, you can select such a predefined map theme, but you cannot modify it because this theme is part of the base map.

The base map becomes the background on which you build interactive themes of information using the geographic map component. You can create as many themes as you wish, but you must define at least one map theme.

Geographic maps have the following data requirements:

- Map configuration requirements:
 - Map Viewer URL: You must provide a URL for the location of the Oracle Application Server MapViewer service. This service is required to run the base map that provides the background for the layers in the ADF geographic map component. OracleAS MapViewer is a programmable tool for rendering maps using spatial data managed by Oracle Spatial. The URL is:
`http://elocation.oracle.com/mapviewer`
 - Geocoder URL: If you want to convert street addresses into coordinates, then you must provide the URL for the Geocoder for the geographic map. A Geocoder is a Web service that converts street addresses into longitude and latitude coordinates for mapping. The URL is:
`http://elocation.oracle.com/geocoder/gcserver`
- Base map: You must have a base map created by the Map Builder tool in OracleAS MapViewer. This base map must define polygons at the level of detail that you require for your map themes to function as desired. For example, if you have a map pie graph or bar graph theme that you want to use for creating graphs in each state of a certain region, then you must have a base map that defines polygons for all these states at some zoom level. You can display only one graph in a polygon.
- Map themes: Each map theme must be bound to a data collection. The data collection must contain location information that can be bound to similar information in the base map.

Configuring Geographic Map Components

The geometric map has parent components, map child components, and components that modify map themes. The prefix `dvt:` occurs at the beginning of each map component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library. You can configure the following map components:

- Map (`map`): The main map component. Unlike other data visualization components, the map component is *not* bound to data. Instead, all the map theme child components are bound individually to data collections. The map component contains general information about the map including the identification of the base map, the URL for the remote server that is running Oracle Application Server MapViewer service and the URL for the Geocoder Web service that converts street addresses into longitude and latitude coordinates for mapping.

The map component supports the following map child components:

- Color theme (`mapColorTheme`): Map layer that you bind to a data collection. The color theme can be used to identify regions on a base maps.
- Point theme (`mapPointTheme`): Map layer that you bind to a data collection. The point theme identifies individual locations on a map.

Optionally, you can use child map point style components (`mapPointStyleItem`) if you want to customize the image that represents points that fall in a certain data value range. To define multiple images, create a component for each image and specify the associated data value range and image.

- Bar graph theme (`mapBarGraphTheme`): Map layer that you bind to a data collection. This theme displays a bar graph at points to represent multiple data values related to that location. For example, this tag might be used to display a graph that shows inventory levels at warehouse locations.

Optionally, use the map bar graph series set component (`mapBarSeriesSet`) to wrap map bar graph series components (`mapBarSeriesItem`) if you want to customize the color of the bars in a map bar graph. Each map bar graph component customizes the color of one bar in a map bar graph.

- Pie graph theme (`mapPieGraphTheme`): Map layer that you bind to a data collection. This theme displays a pie graph at specific points to represent multiple values at that location. For example, this tag might be used to display a graph that shows inventory levels at warehouse locations.

Optionally, use the map pie slice set component (`mapPieSliceSet`) to wrap map pie slice components (`mapPieSliceItem`) if you want to customize the color of the slices in a map pie graph. Each map pie slice component customizes the color of one slice in a map pie graph.

- Predefined graph theme (`predefinedTheme`): Map layer defined using the Map administrator tool stored along with the map metadata in the database. The predefined theme tag is used when you have your own custom Oracle AS MapViewer instance and need to display large datasets that can be rendered directly by the map viewer.
 - Map legend (`mapLegend`): Created automatically when you create a map. Use this component to customize the map legend.
 - Overview map (`mapOverview`): Created automatically when you create a map. Use this tag to customize the overview map that appears in the lower right-hand corner of the map.
- Toolbar (`mapToolbar`): A parent component that allows the map toolbar to be placed in any location on a JSF page that contains a map. This toolbar contains a `mapID` attribute that points to the map associated with the toolbar. The toolbar lets you perform significant interaction with the map at runtime including the ability to display the map legend and to perform selection and distance measurement. The map toolbar tag has no child components.

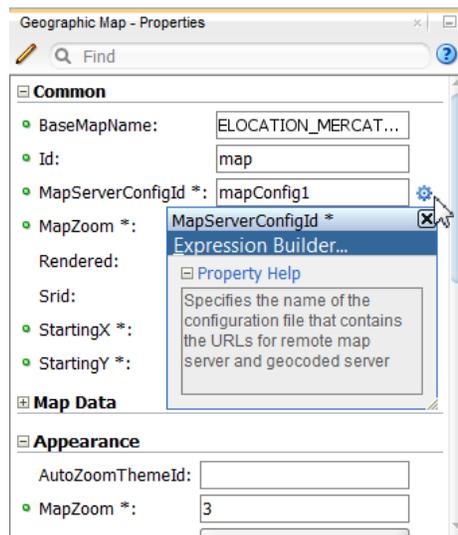
How to Add a Geographic Map to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a geographic map to a JSF page. When you drag and drop a geographic map component onto the page, you are prompted to configure the an Oracle MapViewer service, and optionally a geocoder service.

Once you complete the configuration, and the geographic map is added to your page, you can use the Properties window to configure additional display attributes for the map.

In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. [Figure 30-15](#) shows the dropdown menu for a thematic map component `mapServerConfigId` attribute.

Figure 30-15 Geographic Map `MapServerConfigId` Attribute Dropdown Menu



Note:

If your application uses the Fusion technology stack, then you can use data controls to create a geographic map and the binding will be done for you. For more information, see the "Creating Databound Geographic Maps" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how geographic map attributes and geographic map child components can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Map Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a geographic map to a page:

1. In the ADF Data Visualizations page of the Components window, from the Map panel, drag and drop a **Geographic map** onto the page to open the Create Geographic Map dialog. Use the dialog Maps page to specify **Map Configuration** in one of two ways:
 - Use the dropdown list to choose an available configuration, or
 - Click the Add icon to open the Create Geographic Map Configuration dialog.

In the dialog you can specify the **MapView URL** and **Geocoder URL** from the respective dropdown list, or click the **Add icon** to open the Create URL Connection dialog for each URL.

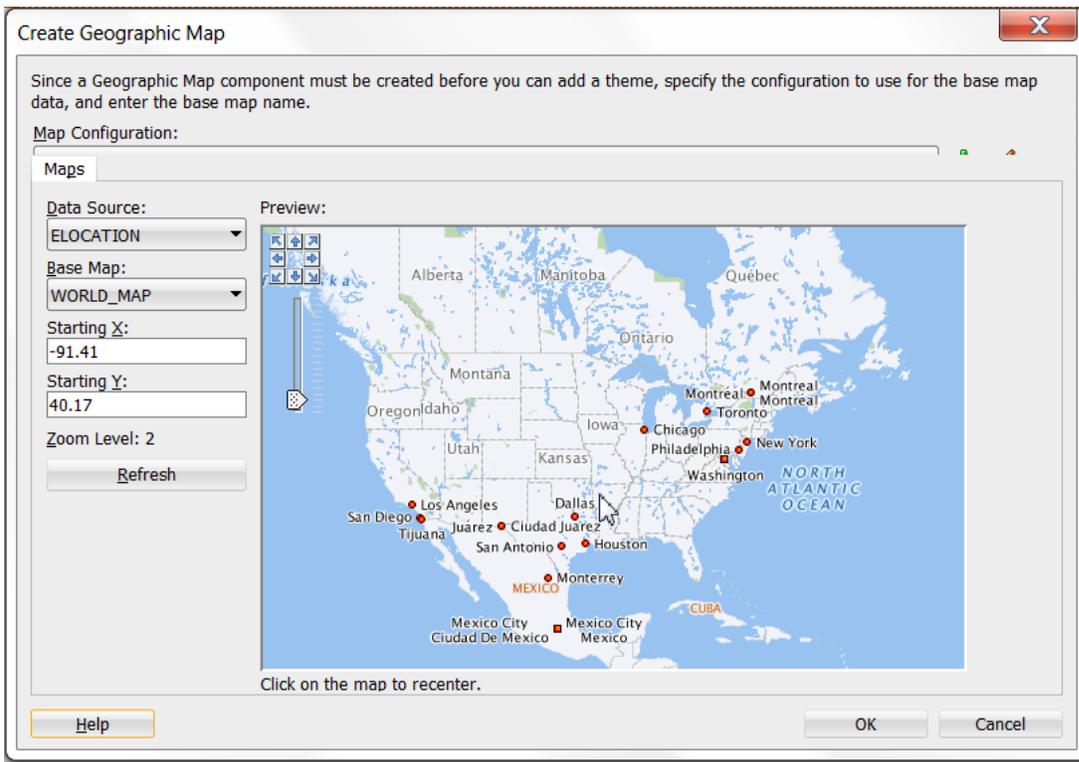
 **Note:**

For the Oracle AS MapViewer service use this URL: `http://elocation.oracle.com/mapviewer`

For the Oracle AS Geocoder service use this URL: `http://elocation.oracle.com/geocoder/gcserver`

For help with the dialog, press F1 or click **Help**. As you complete each dialog to create the configuration, the settings are reflected in the Create Geographic Map dialog. [Figure 30-16](#) shows a sample completed dialog.

Figure 30-16 Maps Page of the Create Geographic Map Dialog



2. Optionally, use the map **Preview** control panel to make adjustments to the center and zoom level of the base map. When you click **Refresh**, the **Starting X** (X coordinate of the center), **Starting Y** (Y coordinate of the center), and **Zoom Level** (initial zoom level) values are updated. If you need help, press F1 or click **Help**.

 **Note:**

Optionally, use the **Themes** page of the Create Geographic Map dialog to add and configure color, point, bar graph, or pie graph themes to display data on the map.

When you create a geographic map using the Data Controls panel and the theme binding dialogs, the data bindings are created for you. For more information, see the "Creating Databound Geographic and Thematic Maps" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

3. In the Properties window, view the attributes for the geographic map. Use the help button to display the complete tag documentation for the `map` component.
4. Expand the **Appearance** section. Use this section to set the following attributes:
 - **ShowScaleBar**: Use to specify the display of the map scale bar. The default value is **false**.
 - **ZoomBarPosition**: Use to specify the location of the map zoom bar. The default value is **West** for placement on the left side of the map. Other valid values include **East** for placement on the right side of the map, and **none** for no zoom bar display.

- **AutoZoomThemeld**: Use to specify the id of the theme where the map view and zoom level will be centered upon initial map display. The value set in the **AutoZoomStrategy** will be used to determine how the map adjusts the center and zoom level.
 - **ShowInfoArea**: Use to specify the display of the information area. The default value is **true**.
 - **MapZoom**: Use to specify the initial zoom level of the geographic map. The zoom levels are defined in the map cache instance as part of the base map.
 - **Unit**: Use to specify the unit of measurement to use for the geographic map. The default value is **miles**. The attribute can also be set to **meters**.
 - **Selection** subsection: Use the attributes in this subsection to define the appearance of the area (rectangular, circular, polygon) and point selection tools. For more information, see [How to Customize and Use Map Selections](#).
5. Expand the **Other** section. Use this section to set the following attributes:
- **AutoZoomStrategy**: Use to specify how the map adjusts the center and zoom level. If the **AutoZoomStrategy** value is set to **MAXZOOM** (default), the map will zoom to the maximum level where all objects in **AutoZoomThemeld** are visible. If the **AutoZoomStrategy** is set to **CENTERATZOOMLEVEL**, the map will center on the theme in **AutoZoomThemeld**, and will use the value in the **MapZoom** attribute as the starting zoom level.
 - **Summary**: Enter a description of the geographic map. This description is accessed by screen reader users.

What Happens When You Add a Geographic Map to a Page

When you use a Components window to add a geographic map to a page, JDeveloper adds code to the JSF page. The example below shows the code added to the JSF page.

```
<dvt:map startingY="46.06" startingX="-78.67" mapZoom="1"
        mapServerConfigId="mapConfig2"
        baseMapName="ELOCATION_MERCATOR.WORLD_MAP"
        inlineStyle="width:600px; height:375px;" id="m2">
  <f:facet name="rtPopup"/>
  <f:facet name="popup"/>
</dvt:map>
```

Note:

JDeveloper automatically inserts two popup facets. The `rtPopup` facet supports a single child component for a right-click `af:menu`. The `popup` facet supports a single child component for a left click `af:dialog` or `af:noteWindow`. For more information about configuring popup components, see [Using Popup Dialogs, Menus, and Windows](#).

You can then configure the geographic map to display data in color, point, pie graph or bar graph themes using the ADF Data Controls panel and the theme binding dialogs. For information about configuring geographic maps to display data, see the "Creating

Databound Geographic and Thematic Maps" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

What You May Need to Know About Active Data Support for Map Point Themes

Geographic map point themes support Active Data Support (ADS) by sending a Partial Page Refresh (PPR) request when an active data event is received.

The PPR response updates the point theme values as follows:

- For update events, the point will update to the new values. If there is a latitude/longitude change, the point will animate to its new location.
- For removal events, the point will be removed from the point theme.
- For insert events, a new point corresponding to the latitude/longitude sent in the change data will be created and visible on the base map.

For additional information about using the ADS, see [Using the Active Data Service with an Asynchronous Backend](#).

Customizing Geographic Map Display Attributes

You can customize the geographic display attributes of the ADF DVT Geographic Map component, such as the size of the map, how the map centers and zoom the map size, zoom strategy, the appearance of selected regions, and the display of the map legend.

How to Adjust the Map Size

You can control the width and height of the map by using the `inlineStyle` attribute in the `dvt:map` tag.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

To adjust the size of a map:

1. In the Structure window, right-click the **dvt:map** component and choose **Go to Properties**.
2. In Properties window, expand the **Style** section. Specify the initial size of the map in the **InlineStyle** attribute.

For example, to specify a width of 600 pixels and a height of 400 pixels, use the following setting:

```
width:600px;height:400px
```

For a map that uses half the available width and height of the page, use the following setting:

```
width:50%;height:50%
```

Best Practice Tip:

Instead of specifying width at 100% in the `inlineStyle` attribute, set the `styleClass` attribute to `AFStretchWidth`.

How to Specify Strategy for Map Zoom Control

You can customize how the geographic map display attributes for the initial zoom level, starting location, initial map theme, and zoom strategy.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured at least one map color, point, pie graph, or bar graph theme to display data on the map.

To control the initial zoom and starting location on a map:

1. In the Structure window, right-click the **dvt:map** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. Use this section to set the following attributes:
 - **AutoZoomThemeID**: Enter the Id of the first theme that will be displayed.
 - **ZoomBarStrategy**: Select the default value **MAXZOOM** to direct the map to zoom down to the maximum level where all objects in the **AutoZoomThemeId** are visible, or select **CENTERATZOOMLEVEL** to direct the map to center on the theme in **AutoZoomThemeId** and to set the zoom level to the value in the **MapZoom** attribute.
 - If you want to change the starting location on the map, enter latitude and longitude in **StartingX** and **StartingY** respectively.
 - **MapZoom**: Enter the beginning zoom level for the map. This setting is required for the zoom bar strategy **CENTERATZOOMLEVEL**.

Note:

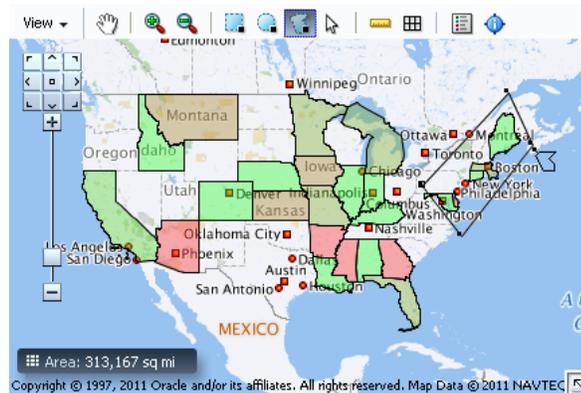
The property `autoZoomThemeID` takes precedence over the property set in `mapZoom`.

How to Customize and Use Map Selections

The geographic map provides selection tools for map areas displaying data in color, pie graph, and bar graph themes. By default, the selection tools are available on the map toolbar and include options for rectangular, circular, polygonal, or point selection.

Figure 30-17 show the use of the polygon tool to select an area in a map with a color theme.

Figure 30-17 Polygon Selection Tool in Map



You can customize the attributes used by the selection tools when users make selection on a `colorTheme`, `barGraphTheme`, and `pieGraphTheme`, using the rectangle tool, circle tool, polygon tool or point tool on the `mapToolbar`.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured at least one map color, point, pie graph, or bar graph theme to display data on the map.

To customize map selection tool attributes:

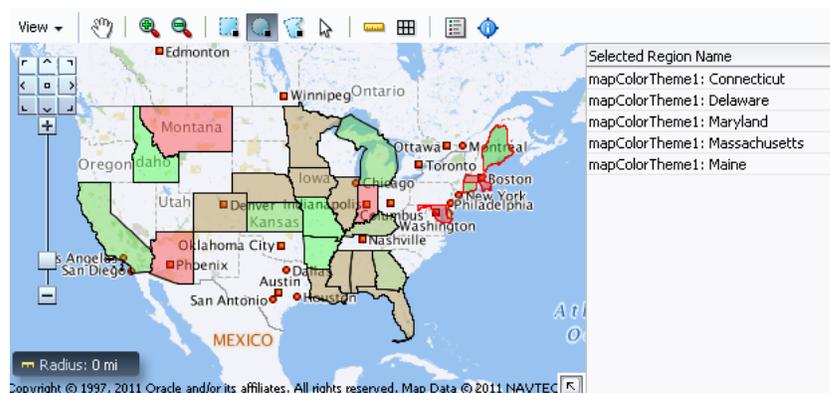
1. In the Structure window, right-click the `dvt:map` component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. Use this section to set the following attributes:
 - **SelectionFillColor**: Use to specify the fill color for the selected region. Valid values are RGB hexadecimal. For example, `color="#000000"` for black. The default value is **#49E0F6**.

- **SelectionStrokeColor:** Use to specify the stroke color for the selected region. Valid values are RGB hexadecimal. For example, `color="#000000"` for black, the default color.
- **SelectionOpacity:** Use to specify the opacity of the fill color for the selected region. Valid values range from **0** to **100**, where **0** is 100% transparent, and **100** is opaque. The default value is **40**.

When an end user clicks on a selection tool and uses the tool to highlight an area on the map, the data values in that area can be displayed in another UI element such as a table, or totalled in the information panel, by using a selection listener.

Figure 30-18 shows a map with the states selected in the area outlined in the color red. The related information about the area is displayed in an associated table.

Figure 30-18 Selected Map Area with Table Data Display



You can provide a selection listener that totals the values associated with a map area selected with one of the map selection tools such as the rectangular selection tool. The total is displayed in an area under the map. You must provide a class that takes `MapSelectionEvent` as an argument in a backing bean method. The example below shows sample code for a backing bean.

```
package view;
import java.util.Iterator;
import oracle.adf.view.faces.bi.component.geoMap.DataContent;
import oracle.adf.view.faces.bi.event.MapSelectionEvent;
public class SelectionListener {
    private double m_total = 0.0;
    public SelectionListener() {
    }
    public void processSelection(MapSelectionEvent mapSelectionEvent) {
        // Add event code here...
        m_total = 0.0;
        Iterator selIterator = mapSelectionEvent.getIterator();
        while (selIterator.hasNext())
        {
            DataContent dataContent = (DataContent) selIterator.next();
            if (dataContent.getValues() != null)
            {
                Double allData[] = dataContent.getValues();
            }
        }
    }
}
```

```

        m_total += allData[0];
    }
}
}
public double getTotal () {
    return m_total;
}
public void setTotal (double total) {
    m_total = total;
}
}
}

```

To provide a selection listener to total map selection values:

1. In the Structure window, right-click the **dvt:map** component and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section. For the **SelectionListener** attribute, enter a method reference that points to the backing bean. For example:

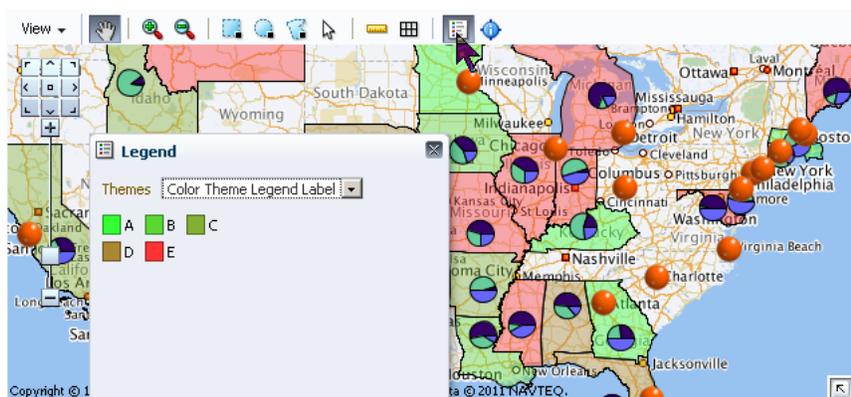
```
#{eventBean.processSelection}
```

How to Customize the Map Legend

The map legend provides an explanation of the map theme data in symbol and label pairs. The legend displays by default in a popup upon initial display of the map, and when a map toolbar is added and configured, the map legend can display in a window when the user clicks the legend toolbar button.

When multiple themes are configured, a dropdown list is available to display the legend for each theme. [Figure 30-19](#) shows a map displaying the legend for one of its multiple themes.

Figure 30-19 Geographic Map Theme Legend Display



 **Note:**

In order for the legend toolbar button to be available, you must add and configure a toolbar to the map. For more information, see [Adding a Toolbar to a Geographic Map](#).

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured at least one map color, point, pie graph, or bar graph theme to display data on the map.

To customize a map legend:

1. In the Structure window, right-click the **dvt:map** component and choose **Insert Inside Geographic Map > Legend**.
2. In the Properties window, expand the **Common** section. In this section set the following attributes:
 - **InitialShown**: Use to specify whether or not the map legend is displayed in a popup upon initial display of the map. The default value is **true**.
 - **Width**: Enter the width of the legend. The default value is **200px**.
 - **Height**: Enter the height of the legend. The default value is **150px**.
 - **PointThemeLabel**: Enter the label to use for the map point theme in the legend. All point themes are shown as one option in the dropdown list. The default value is **Point Theme**.
 - **NumberOfColumns**: Enter the number of columns to use for displaying the colors of a color map theme in the legend.

For example, if a color theme has 15 colors, and the value of the attribute is set to **3**, then the legend will show 5 rows on the legend for the theme, where each row has 3 columns.

What You May Need to Know About Skinning and Customizing the Appearance of Geographic Maps

For the complete list of geographic map skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Customizing Geographic Map Themes

Each of the ADF DVT Geographic Map themes, color, point, pie chart, and bar chart, can be customized using one or more of the following: the map theme binding dialogs, the attributes of the theme tag, or the child tags of the theme tag.

How to Customize Zoom Levels for a Theme

For all map themes, you must verify that the theme specifies zoom levels that match the related zoom levels in the base map. For example, if the base map shows counties only at zoom levels 6 through 8, then a theme that displays points or graphs by county should be applied only at zoom levels 6 through 8.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured at least one map color, point, pie graph, or bar graph theme to display data on the map.

To customize the zoom levels of a map theme:

1. In the Structure window, right-click the **dvt:mapColorTheme**, **dvt:mapPointTheme**, **dvt:mapBarGraphTheme**, or **dvt:mapPieGraphTheme** component that you want to customize, and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. For the **MinZoom** and **MaxZoom** attribute, enter the desired low and high zoom values, respectively.

How to Customize the Labels of a Map Theme

By default, the `Id` attribute of a map theme is used as the label when that theme is displayed in the legend or in the **View** menu, Theme Selection dialog. You can customize map theme labels using `shortLabel` and `menuLabel` attributes to create meaningful labels that identify both the theme type (color, point, bar graph, or pie graph) and the data (such as population, sales, or inventory) so that users can easily recognize the available themes.

Use these attributes to create meaningful labels that identify both the theme type (color, point, bar graph, or pie graph) and the data (such as population, sales, or inventory) so that users can easily recognize the available themes.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured at least one map color, point, pie graph, or bar graph theme to display data on the map.

To customize the labels of a map theme:

1. In the Structure window, right-click the **dvt:mapColorTheme**, **dvt:mapPointTheme**, **dvt:mapBarGraphTheme**, or **dvt:mapPieGraphTheme** component that you want to customize, and choose **Go to Properties**.
2. In the Properties window expand the **Appearance** section. Use this section to set the following attributes:
 - **ShortLabel**: Use to specify a label for the theme when displayed in the map legend.
 - **MenuLabel**: Use to specify a label for the theme in the **View** menu, Theme Selection dialog.

For example, you might want to enter the following text for a color theme that colors New England states according to population:

```
shortLabel="Color - Population, NE Region"
```

How to Customize Color Map Themes

When you create a color map theme, you can customize the colors used for the coloring of the background layer. You can specify the colors associated with the minimum and maximum ranges, and then specify the number of color ranges for the theme. For example, if the colors relate to the population on the map, the least populated areas display the minimum color and the most populated areas display the maximum color. Graduated colors between the minimum and maximum color are displayed for ranges between these values.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured a map color theme to display data on the map.

To customize the colors of a color map theme:

1. In the Structure window, right-click the **dvt:mapColorTheme** component and choose **Go to Properties**.
2. In the Properties window, expand the **Theme Data** section. Use this section to set the following attributes:
 - If you want to change the default colors associated with the minimum and maximum range of data values, then select the desired colors for the **MinColor** and **MaxColor** attributes respectively.

- If you want to change the default number of color ranges for this theme, change the integer in the **BucketCount** attribute.

For example, if `<dvt:mapColorTheme minColor="#000000" maxColor="#ffffff" bucketCount="5"/>`, then the colors for the five buckets are: #000000, #444444, #888888, #bbbbbb, #ffffff.

Alternatively, you can specify the color for each bucket. To specify colors for multiple buckets, for the **ColorList** attribute of `mapColorTheme`, bind a color array to the attribute or use a semicolon-separated string. Color can be specified using RGB hexadecimal.

For example, if the value is `colorList="#ff0000;#00ff00;#0000ff"`, then the value of the first bucket is red, the second bucket is green, and the third bucket is blue.

How to Customize Point Images in a Point Theme

A map point theme uses a default image to identify each point. However, you can specify multiple custom images for a point theme and identify the range of data values that each image should represent, using a `mapPointStyleItem` component for each custom image you want to use in a map point theme.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured a map point theme to display data on the map.

To customize the images for points in a map point theme:

1. In the Structure window, right-click the **dvt:mapPointTheme** component and choose **Insert Inside Map Point Theme > Point Style Item**.
2. In the Properties window, expand the **Common** section. Use this section to set the following attributes:
 - **ImageURL**: Use to specify the path to the image file to display on the map for a point that falls in the data value range for this custom image. Alternatively, you can choose **Edit** from the attribute dropdown menu to open a dialog to navigate to the image file.
 - **MaxValue** and **MinValue**: Use to specify the data value range that this custom image represents by entering minimum values and maximum values for each attribute respectively.
 - **Id**: Enter a unique identifier for the custom image that you are defining.
 - **ShortLabel**: Use to specify the descriptive text that you want to display in front of the actual data value when a user hovers the cursor over a point that falls in the range represented by this tag.

For example, you might want to enter the following text for a custom point that falls in the lowest data value range:

Low Inventory

- **HoverURL:** Optionally, specify the path to a different image to use when the end user moves the mouse over an image on the map.
 - **SelectedURL:** Optionally, specify the path to a different image to use when the end user selects an image on the map.
3. Repeat Step 2 for each custom image that you want to create for your point theme.

What Happens When You Customize the Point Images in a Map

When you use the point style item components to specify a custom image representing a range of data values for a point theme, a child `mapPointStyleItem` tag is defined inside the parent `mapPointTheme` tag.

The initial point style setting (`ps0`) applies to values that do not exceed 500. This point style displays an image for very low inventory and provides corresponding tooltip information.

The second point style setting (`ps1`) applies to values between 500 and 1000. This point style displays an image for low inventory and provides corresponding tooltip information.

The final point style setting (`ps2`) applies to values between 1000 and 1600. This point style displays an image for high inventory and provides corresponding tooltip information.

The example below shows the code generated on a JSF page for a map point theme that has three custom point images that represent ranges of inventory at each warehouse point.

```
<dvt:map id="map1"
...
  <dvt:mapPointTheme id="mapPointTheme1"
    shortLabel="Warehouse Inventory"
    value="{bindings.WarehouseStockLevelsByProduct1.geoMapModel}">
    <dvt:mapPointStyleItem id="ps0" minValue="0"
      maxValue="500"
      imageURL="/images/low.png"
      selectedImageURL="/images/lowSelected.png"
      shortLabel="Very Low Inventory"/>
    <dvt:mapPointStyleItem id="ps1" minValue="500"
      maxValue="1000"
      imageURL="/images/medium.png"
      selectedImageURL="/images/mediumSelected.png"
      shortLabel="Low Inventory"/>
    <dvt:mapPointStyleItem id="ps2" minValue="1000"
      maxValue="1600"
      imageURL="/images/regularGreen.png"
      selectedImageURL="/images/regularGreenSelected.png"
      shortLabel="High Inventory"/>
  </dvt:mapPointTheme>
</dvt:map>
```

How to Customize the Bars in a Bar Graph Theme

When you create a map bar graph theme, default colors are assigned to the bars in the graph. You can customize the colors of the bars. Use one `mapBarSeriesSet` tag to wrap all the `mapBarSeriesItem` tags for a bar graph theme and insert a `mapBarSeriesItem` tag for each bar in the graph.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured a map bar graph theme to display data on the map.

To customize the color of the bars in a map bar graph theme:

1. In the Structure window, right-click the **dvt:mapBarGraphTheme** tag and choose **Insert Inside Bar Graph Theme > Map Bar Series Set**.
There are no attributes to set for this tag. It is used to wrap the individual bar series item tags.
2. In the Structure window, right-click the **dvt:mapBarSeriesSet** tag and choose **Insert Inside Bar Series Set > Map Bar Series Item**.
3. In the Properties window, set the following attributes:
 - **Id**: Enter a unique Id for the bar series item.
 - **Color**: Enter the unique color to use for the bar. Valid values are RGB hexadecimal colors. Alternatively, you can choose **Edit** from the attribute dropdown menu to open an Edit Property: Color dialog.
4. Repeat Step 3 for each bar in the graph.

 **Note:**

To find and modify the sequence of the bars in the graph, examine the Edit Bar Graph Map Theme Binding dialog by clicking the **Edit icon** for the `mapBarGraphTheme` component. The sequence of the entries in the **Series Attribute** column of that dialog determines the sequence that bars appear in the graph. After selecting an existing series, use the arrow icons (**Up**, **Down**, **Top**, **Bottom**) to reorder the series or use the **Delete** icon to delete that series.

What Happens When You Customize the Bars in a Map Bar Graph Theme

When you use the Edit Bar Graph Map Theme Binding dialog to customize the bars in a map bar graph theme, the sequence of the bars reflect the sequence of the entries in the **Series Attribute** column in the dialog.

The example below shows sample source code generated on the JSF page when you customize the bars in a map bar graph.

```
<dvt:map
...
  <dvt:mapBarGraphTheme
...
    <dvt:mapBarSeriesSet>
      <dvt:mapBarSeriesItem color="#333399" id="bar1"/>
      <dvt:mapBarSeriesItem color="#0000ff" id="bar2"/>
    </dvt:mapBarSeriesSet>
  </dvt:mapBarGraphTheme>
</dvt:map>
```

How to Customize the Slices in a Pie Graph Theme

When you create a map pie graph theme, default colors are assigned to the slices in the graph. You can customize the colors of the slices. Use one `mapPieSliceSet` tag to wrap all the `mapPieSliceItem` tags for a pie graph theme and insert a `mapPieSliceItem` tag for each slice in the graph.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

You should have already configured a map pie graph theme to display data on the map.

To customize the color of the slices in a map pie graph theme:

1. In the Structure window, right-click the **dvt:mapPieGraphTheme** tag and choose **Insert Inside Pie Graph Theme > Pie Slice Set**.

There are no attributes to set for this tag. It is used to wrap the individual pie graph item tags.

2. In the Structure window, right-click the **dvt:mapPieSliceSet** node and choose **Insert Inside Map Pie Slice Set > Pie Slice Item**.

3. In the Properties window, set the following attributes:

- **Id**: Enter a unique Id for the pie slice item.

- **Color:** Enter the unique color to use for the pie slice. Valid values are RGB hexadecimal colors. Alternatively, you can choose **Edit** from the attribute dropdown menu to open an Edit Property: Color dialog.
4. Repeat Step 3 for each pie slice in the graph.

 **Note:**

To find and modify the sequence of the slices in the graph, examine the Edit Pie Graph Map Theme Binding dialog by clicking the **Edit icon** for the `mapPieGraphTheme` component. The sequence of the entries in the **Pie Slice Attribute** column of that dialog determines the sequence that bars appear in the graph. After selecting an existing pie slice, use the arrow icons (**Up, Down, Top, Bottom**) to reorder the slices or use the **Delete** icon to delete that slice.

What Happens When You Customize the Slices in a Map Pie Graph Theme

When you use the Edit Pie Graph Map Theme Binding dialog to customize the slices in a map pie graph theme, the sequence of the slices reflect the sequence of the entries in the **Pie Slices Attribute** column of the dialog.

The example below shows sample code generated in a JSF page when you customize the slices in a map pie graph.

```
<dvt:map
...
  <dvt:mapPieGraphTheme
...
    <dvt:mapPieSliceSet>
      <dvt:mapPieSliceItem color="#ffffff" id="slice1"/>
      <dvt:mapPieSliceItem color="#ffff00" id="slice2"/>
      <dvt:mapPieSliceItem color="#ff0000" id="slice3"/>
    </dvt:mapPieSliceSet>
  </dvt:mapPieGraphTheme>
</dvt:map>
```

Adding a Toolbar to a Geographic Map

When you create an ADF DVT Geographic Map, you can also add and configure a map toolbar to display the legend and information panel, select themes (if you have multiple themes of the same type) or use any of the distance measurement, area measurement, or selection tools.

Figure 30-20 shows a map toolbar.

Figure 30-20 Geographic Map Toolbar



For more information about toolbar functionality, see [Geographic Map End User and Presentation Features](#).

How to Add a Toolbar to a Map

The map toolbar is a separate component and can be positioned on the JSF page above or below the map.

Before you begin:

It may be helpful to have an understanding of how map attributes and map child tags can affect functionality. For more information, see [Configuring Geographic Map Components](#).

You should already have a map on your page. If you do not, follow the instructions in this chapter to create a map. For information, see [How to Add a Geographic Map to a Page](#).

To add and configure a map toolbar:

1. In the Structure window, right-click the **dvt:map** component and choose **Insert before Geographic Map** or **Insert after Geographic Map > ADF Data Visualization** to open the **ADF Data Visualization Item** dialog.
2. Use the dialog to select **Toolbar** to open the Create Map Toolbar dialog.
3. From the dialog dropdown list, choose the ID of the map on which this toolbar will operate and click **OK**.
4. In the Properties window, expand the **Common** section. In this section set the following attributes:
 - **ShowDistanceTools**: Use to specify whether or not the distance tool is available on the toolbar. The default value is **true**.
 - **ShowSelectThemeDialog**: Use to specify whether or not the Select Theme dialog is available on the **View** menu of the toolbar. The default value is **true**.
 - **ShowSelectThemeMenuItem**: Use to specify whether or not the Select Theme option is available on the **View** menu of the toolbar. The default value is **true**.
 - **ShowSelectionTools**: Use to specify whether or not the selection tools, area rectangle, circle, polygon, or point tool is available on the toolbar. The default value is **true**.
 - **ShowViewMenu**: Use to specify whether or not the **View** menu is available on the toolbar. The default is **true**.
 - **ShowZoomTools**: Use to specify whether or not the zoom in and zoom out tools are available on the toolbar. The default is **true**.

What Happens When You Add a Toolbar to a Map

When you add a toolbar to a map, the following occurs:

- A toolbar appears in the JSF page above or below the map as specified. By default, the toolbar contains all the tools unless you change the visibility of one or more tools.

- Source code is generated and appears in the JSF page above or below the code for the map.

The example below shows sample code for a toolbar that is associated with a map with the ID of `map_us`. The example shows the location of the code for the map.

```
<af:form>
  <dvt:mapToolbar mapId="map_us" id="T1"/>
  <dvt:map id="map_us"
    ...
  </dvt:map>
</af:form>
```

Using Thematic Map Components

To display data in ADF DVT thematic maps, a named data collection is required. A data collection represents a set of data objects (also known as a row set) in the data model. Each object in a data collection represents a specific structured data item (also known as a row) in the data model.

Configuring Thematic Maps

The thematic map has a parent component that specifies the geographic base map and child components that are used to style map regions with colors, patterns, or markers, or both, or to add a legend to the map. The prefix `dvt:` occurs at the beginning of each thematic map component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure the following map components:

- Thematic map component (`thematicMap`): The main thematic map component used to specify the base map upon which data is displayed. The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, as well as base maps for continents and regions of the world such as EMEA and APAC. The thematic map component does not require a map service to display a base map.
- Layer (`areaLayer`): Use to specify the layers in the base map that are displayed. Each `areaLayer` component references a single layer, for example, **Country**, or **States** in the **USA** base map, and only the map layers for which an `areaLayer` tag is present will be displayed. Data is then associated with a layer by nesting a data layer within the layer. The `areaLayer` child tags are area data layer (`areaDataLayer`) and point data layer (`pointDataLayer`).
- Area Data Layer (`areaDataLayer`): Use to associate map layers with a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image.

 **Note:**

When you use stamping, child components are not created for every area, marker, or image in a thematic map. Rather, the content of the component is repeatedly rendered, or stamped, once per data attribute, such as the rows in a data collection.

Each time a child component is stamped, the data for the current component is copied into a `var` property used by the data layer component in an EL Expression. Once the thematic map has completed rendering, the `var` property is removed, or reverted back to its previous value.

The location of the data layer is identified by its immediate child, the `areaLocation` tag. This component specifies the location of the named region or area in the map layer. The three types of data that can be stylized in a child tag to the `areaLocation` tag include:

- Area (`area`): Use to stamp out stylistic attributes such as fill colors, patterns, or opacity onto the geographical regions of the map.
- Marker (`marker`): Use to stamp out built-in or custom shapes associated with data points on the map. Markers can be customized with different stylistic attributes such as colors and patterns based on their underlying data.
- Images (`af:image`): Use to stamp out an image associated with geographical regions of the map.

 **Note:**

Instead of directly specifying the style attributes for the `area` or `marker` tags, you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical definitions of the data set. If the same style attribute is set in both the `area` or `marker` tags, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

 **Note:**

You can format a numerical value represented in the `area` or `marker` tag, for example, apply a currency format, by using an `af:convertNumber` tag. For more information, see [How to Format Numeric Data Values in Area and Marker Labels](#).

The basic tag structure for configuring a data layer in a thematic map is illustrated as follows:

```
<dvt:thematicMap basemap="usa"...>
  <dvt:areaLayer layer="states">
```

```

<dvt:areaDataLayer value="{mydata.collectionModel}" var="row">
  <dvt:areaLocation name="{row.State}">
    <dvt:area> OR <dvt:marker> OR <af:image>
      <dvt:attributeGroups> OR <af:convertNumber>
    </dvt:attributeGroups> OR </af:convertNumber>
    </dvt:area> OR </dvt:marker> OR </af:image>
  </dvt:areaLocation>
</dvt:areaDataLayer>
</dvt:areaLayer>
</dvt:areaLayer>
<dvt:areaLayer layer="counties"/>
</dvt:thematicMap>

```

- Point data layer (`pointDataLayer`): Use to associate a map with a specific data point or a map layer with a data collection. The data point can be specified by a named point in a map layer, for example, cities in the USA map, or by longitude and latitude. Using stamping, each point in the data model can be identified by a marker, for example a circle or square, or an image.

 **Note:**

When you use stamping, child components are not created for every marker or image in a thematic map. Rather, the content of the component is repeatedly rendered, or stamped, once per data attribute, such as the rows in a data collection.

Each time a child component is stamped, the data for the current component is copied into a `var` property used by the point layer component in an EL Expression. Once the thematic map has completed rendering, the `var` property is removed, or reverted back to its previous value.

The location of the point layer is identified in its immediate child, a `pointLocation` tag. You can configure the location to specify longitude and latitude, or by the location of the named area in the map layer. The two types of data that can be stylized in a child tag to the `pointLocation` tag include:

- Marker (`marker`): Use to stamp out built-in or custom shapes associated with data points on the map. Markers can be customized with different stylistic attributes such as colors and patterns based on their underlying data.
- Images (`af:image`): Use to stamp out an image associated with geographical regions of the map.

 **Note:**

Instead of directly specifying the style attributes for the `marker` tag, you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical definitions of the data set. If the same style attribute is set in both the `marker` tags, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

 **Note:**

You can format a numerical value represented in the `marker` tag, for example, apply a currency format, by using an `af:convertNumber` tag. For more information, see [How to Format Numeric Data Values in Area and Marker Labels](#).

The basic tag structure for configuring a point data layer in a thematic map is illustrated as follows:

```
<dvt:thematicMap basemap="usa"...>
  <dvt:areaLayer layer="states">
    <dvt:pointDataLayer id="pd1" value="{bean.pointData" var="row">
      <dvt:pointLocation type="pointXY"
        pointX="{row.longitude}"
        pointY="{row.latitude}">
        <dvt:marker> OR <af:image>
          <dvt:attributeGroups> OR <af:convertNumber>
          </dvt:attributeGroups> OR </af:convertNumber>
        </dvt:marker> OR </af:image>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

When a point data layer is configured as a direct child of the thematic map component, the data points are always displayed as a global point layer. If the point layer is nested inside a map layer, the data points are only displayed when that map layer is displayed.

The tag structure for nesting point data layers is illustrated following. In the illustration, point data layer `pd1` is only displayed when the `states` layer is displayed. Point data layer `pd2` is always displayed.

```
<dvt:thematicMap basemap="usa"...>
  <dvt:areaLayer layer="states">
    <dvt:areaDataLayer.../>
    <dvt:pointDataLayer id="pd1" ...>
  </dvt:areaLayer>
  <dvt:areaLayer layer="counties"/>
  <dvt:pointDataLayer id="pd2"...>
</dvt:thematicMap>
```

- **Custom layer (`customAreaLayer`):** Use to create a new map layer from independent region data and insert the newly created layer into the layer hierarchy. The custom layer is created by extending a predefined map layer and aggregating the lower level regions to form the new regions in the custom layer. After defining a custom map layer, it is used in the same way as any other map layer

Use the child `customArea` component to specify the regions from the predefined base map that will be aggregated to form the new area.

- Categorical attributes (`attributeGroups`): Use to generate stylistic attribute values such as colors or shapes based on categorical data values in a data set.

An alternative to configuring a default stamp across all areas or markers in the thematic map, you use an `area` or `marker` component child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `area` or `marker` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

Based on the attribute representing the column in the data model to group by, the `attributeGroups` component can generate style values for each unique value, or group, in the data. The `type` property of the `attributeGroups` tag specifies the type of stylistic attribute for which values are produced. Supported types for `area` components are `color`, `pattern`, and `opacity`. Supported types for `marker` components are `color`, `shape`, `pattern`, `opacity`, `scaleX`, and `scaleY`. These types can be combined in a space-delimited list to generate multiple stylistic properties for each unique data value.

The default style values that are generated are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the ADF skin that can be customized by setting the index-based selectors to the desired values.

To achieve a finer level of detail in the display of data, the grouping rules specified in the `attributeGroups` component can be overridden by two types of rules defined in these child components:

- Matching rule (`attributeMatchRule`): Use to substitute an attribute when the data matches a certain value.
- Exception rule (`attributeExceptionRule`): Use to replace an attribute value with another when a particular boolean condition is met.

Note:

The `attributeMatchRule` and `attributeExceptionRule` tags use a child `f:attribute` tag to define the override values, where the `name` property defines the type of attribute (such as color or shape), and the `value` property defines the override value to use (such as red or square). When the `value` property of the `attributeGroups` tag overrides the value in the `group` property of the `attributeMatchRule`, then the style value in the child `f:attribute` will be assigned to that group of data instead of the next value from the default ramp.

The `f:attribute` tag `name` property must match one of the attributes listed in the `type` of the `attributeGroups` tag grouping rules. If the value provided by an override also happens to be in the prebuilt ramp returned by the `attributeGroups`, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

- Legend (`legend`): Use to display an explanatory table of the map's styled data in symbol and label pairs. Legend components support symbols for color, shape, custom shape, fill pattern, opacity, and images.
 - Legend section (`legendSection`): Use one or more to point to a thematic map `area`, `marker`, `attributeGroups`, or `af:image` components stamped to style

the data displayed in the map. Legend items sourced from the `attributeGroups` component split area or marker attribute types into different sections.

- Legend group (`showLegendGroup`): Use to create a disclosable section that contains legend section components.

Using the Layer Browser

JDeveloper provides a Layer Browser as a tool to ease the development of structuring map layers, data layers, and styling areas and markers to display data in a thematic map. The Layer Browser displays on top of the thematic map in the visual editor and can be repositioned and resized. If not visible, right-click in the map and choose **Open Layer Browser**.

The Layer Browser visually represents the logical structure of the thematic map and its hierarchical map layers and components. Selection of a component in the Layer Browser is coordinated with selection in the Properties window, Structure window, and page source code.

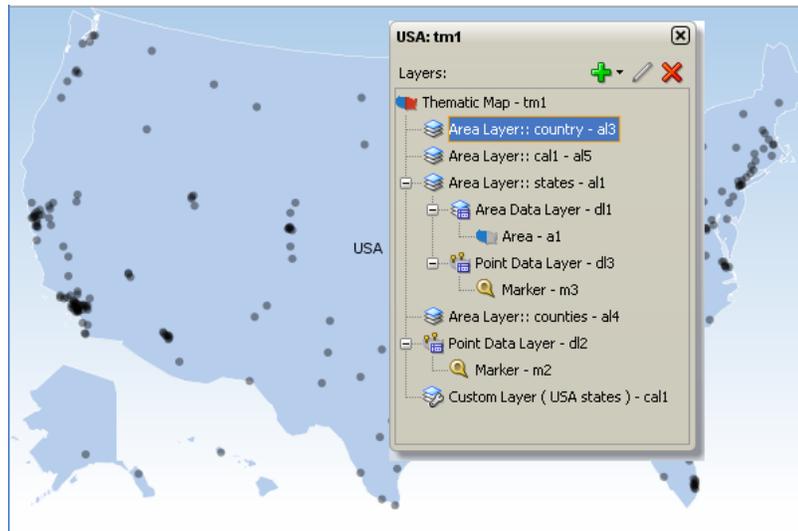
The Layer Browser toolbar provides controls for the following operations:

- Add: Provides a dropdown menu for opening a create map layer, data layer, area, or marker dialog to add and configure the component and add source code to the thematic map. The menu choices are provided to maintain the correct structure of the thematic map
- Edit: Open a data layer, area, or marker binding dialog to modify the settings for the component and change the thematic map source code.
- Delete: Remove a selected map layer, data layer, area or marker from the thematic map structure and source code.

The Layer Browser displays the Id for each component represented in the hierarchical structure. Components are automatically assigned a unique, consecutively numbered id value. Map layers (`areaLayer`) are assigned `a11`, `a12`, `a13`, and so on. Custom layers (`customAreaLayer`) are assigned `ca11`, `ca12`, `ca13`, and then referenced by an `areaLayer` component within the consecutive order. Data layers including area (`areaDataLayer`) and point (`pointDataLayer`) components are assigned `d11`, `d12`, `d13`, and so on. When a point layer (`pointDataLayer`) is added as a direct child of the thematic map, it is a global point layer and always displayed in the thematic map. Markers (`marker`) are assigned `m1`, `m2`, `m3` and so on. Areas (`area`) are assigned `a1`, `a2`, `a3`, and so on.

[Figure 30-21](#) shows a Layer Browser displaying the hierarchical structure of a thematic map.

Figure 30-21 Thematic Map Layer Browser

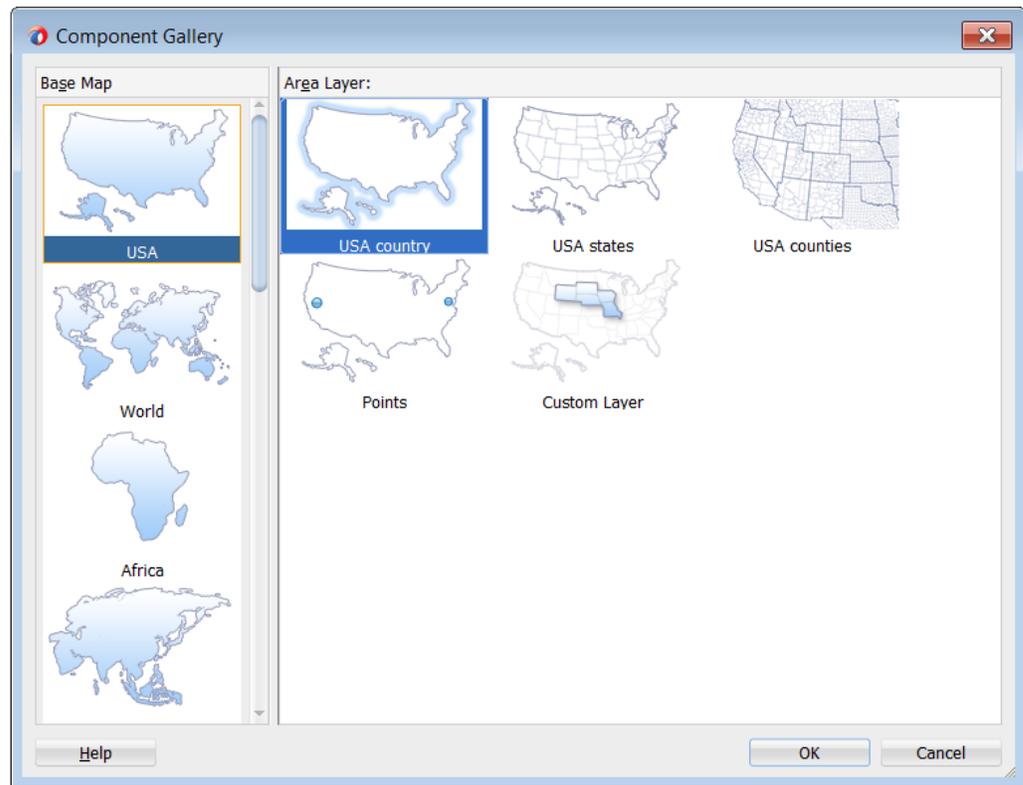


How to Add a Thematic Map to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a thematic map to a JSF page. When you drag and drop a thematic map component onto the page, the Component Gallery displays available base maps, prebuilt regional layers, and a custom layer option to provide visual assistance when creating thematic maps.

[Figure 30-22](#) show the Component Gallery for thematic maps with the United States base map and states layer selected.

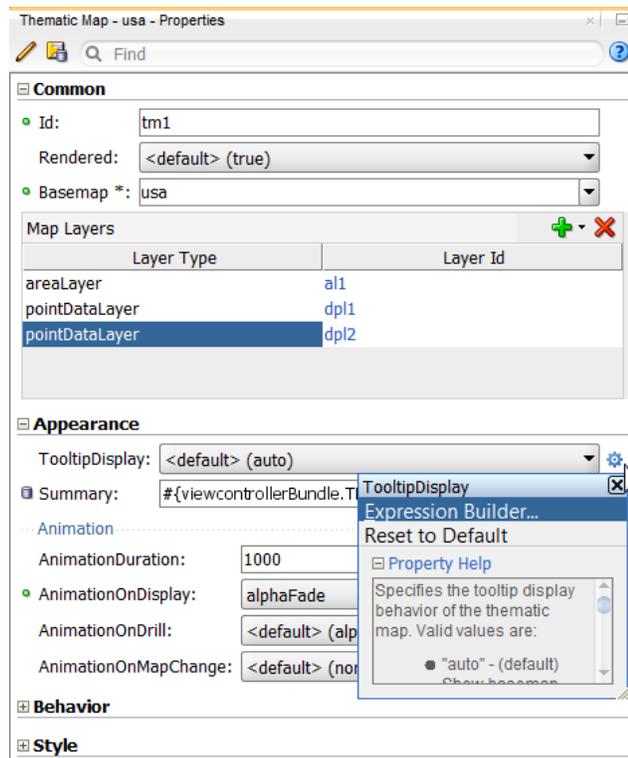
Figure 30-22 Component Gallery for Thematic Maps



Once you complete the dialog, and the thematic map is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the map.

In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. [Figure 30-23](#) shows the dropdown menu for a thematic map component `toolTipDisplay` attribute.

Figure 30-23 Thematic Map ToolTipDisplay Attribute Dropdown Menu



Note:

If your application uses the Fusion technology stack, then you can use data controls to create a thematic map and the binding will be done for you. See the "Creating Databound Thematic Maps" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. See [Configuring Thematic Maps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Map Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

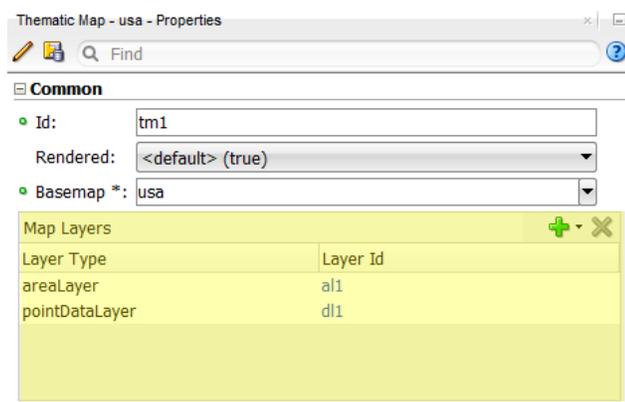
To add a thematic map to a page:

1. In the ADF Data Visualizations page of the Components window, from the Map panel, drag and drop a **Thematic map** onto the page to open the Create Thematic Map dialog in the Component Gallery.

Use the dialog to select the thematic map base map and layer in the prebuilt map hierarchy that you want the thematic map to display. For help with the dialog, press F1 or click **Help**.

2. In the Properties window, view the attributes for the thematic map. Use the help button to display the complete tag documentation for the `thematicMap` component.
3. Expand the **Common** section. Use this section to set the following attributes:
 - **Basemap**: If you want to change the base map selected in the Component Gallery, use the dropdown list to select any of the following valid values: **usa**, **world**, **africa**, **asia**, **australia**, **europa**, **northAmerica**, **southAmerica**, **apac**, **emea**, **latinAmerica**, **usaAndCanada**, or **worldRegions**.
 - **Map Layers**: Use the dialog that displays inside the Properties window to add additional map layers you wish to display in the thematic map. For example, the USA base map includes a map layer for the country, the states, the counties, the cities (points), and a custom layer. Use the dropdown list associated with the **Add icon** to add available map layers in the predefined geographic hierarchy, to create a custom map layer and insert it into the hierarchy, or to add a global point layer into the base map. Use the **Delete icon** to delete a layer you do not wish to display in the thematic map. [Figure 30-24](#) shows the map layers dialog highlighted in the Properties window for the `thematicMap` component.

Figure 30-24 Map Layers Dialog in the Properties Window



4. Expand the **Appearance** section. Use this section to set the following attributes:
 - **TooltipDisplay**: By default (**auto**), thematic maps automatically display tooltips using prebuilt map labels when the user moves the cursor over the map. If data is available, the label is concatenated with the `shortDesc` attribute from the `area` or `marker` component stamp. Other valid values include **none** to disable the display of tooltips, and **shortDesc** to display only the data coming from the stamps, not including the prebuilt label of the base map.
 - **Animation** subsection: Use the animation attributes in this subsection to configure animation in thematic maps.
5. Expand the **Behavior** section. Use this section to set the following attributes:

- **Drilling:** Use to enable drilling the data view between thematic map layers. From the dropdown list select **on** to enable drilling. The default value is **off**.
 - **MaintainDrill:** Use to specify an optional **true** value for maintaining the drilled state of a previously drilled area when a new area is drilled. The default value is **false**.
 - **DrillBehavior:** Use to specify an optional **zoomToFit** effect on the area being drilled. The default value is **none**.
 - **ControlPanelBehavior:** Use the dropdown list to select the display of the thematic map control panel. The default value is **initCollapsed** for only display of the hide/show button. Other valid values include **hidden** and **initExpanded**.
 - **FeaturesOff:** Enter a space delimited list of end user features to disable at runtime. Valid values are **pan**, **zoom**, and **zoomToFit**. The default value is **none**.
 - **MaxZoom:** Enter any number greater than **1.0** to specify the maximum zoom level for the component. A value of **10.0** indicates that the map can be zoomed in until the base map appears at 10x, the viewport zoomed to fit size. The default value is **6.0**.
6. Expand the **Other** category. For the **Summary** attribute, enter a description of the thematic map. This description is accessed by screen reader users.

You can also use the Layer Browser to add map layers to the thematic map. See [Using the Layer Browser](#).

[Customizing Thematic Map Display Attributes](#) describes additional ways of customizing the map.

[How to Configure Animation Effects](#) described how to use animation attributes.

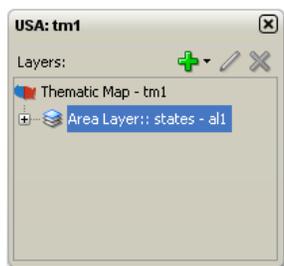
What Happens When You Add a Thematic Map to a Page

When you use the Components window to create a thematic map, JDeveloper inserts code in the JSF page. The example below shows the code inserted in the JSF page.

```
<dvt:thematicMap basemap="usa" id="tm1">
  <dvt:areaLayer layer="states" id="all"/>
</dvt:thematicMap>
```

The Layer Browser displays the hierarchical structure of the thematic map. [Figure 30-25](#) shows the Layer Browser after using the Components window to create a thematic map.

Figure 30-25 Thematic Map Layer Browser



You can then configure the thematic map to display data in stylized areas or markers using the ADF Data Controls panel and the thematic map binding dialogs. For information about configuring thematic maps to display data, see the "Creating Databound Thematic Maps" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

What You May Need to Know About Thematic Map Image Formats

Thematic maps support the following image formats: HTML5, Flash, and Portable Network Graphics (PNG). All image formats support locales with right-to-left display. By default, thematic maps will display in the best output format supported by the client browser.

If the best output format is not available on the client, the application will default to an available format. For example, if the client does not support HTML5, the application will use:

- Flash, if the Flash Player is available

You can control the use of Flash content across the entire application by setting a `flash-player-usage` context parameter in `adf-config.xml`. For more information, see [Configuring Flash as Component Output Format](#).

- PNG output format

Although static rendering, such as maintaining pan and zoom state of the Flash display, is fully supported when using the printable PNG output format, certain interactive features are not available including:

- Animation
- Context menus
- Drag and drop gestures
- Popup support
- Selection

Defining Thematic Map Base Maps

The ADF DVT thematic map component support the use of an extensive set of prebuilt base maps with layers that represent a set of regions. You can also define a custom base map for your thematic map using a set of `mapProvider` APIs or configuring an XML metadata file to identify data points on an image file.

Using Prebuilt Base Maps

Each base map provided for the thematic map component has two or more prebuilt map layers that represent a set of regions. For example, the `world` base map includes a map layer for `continents` and another layer for `countries`.

The regions in the lower level map layers are aggregated to make up the next level in the geographical hierarchy. The map layer is specified in the `layer` attribute of the `areaLayer` component. Each base map includes several sets of regions and one fixed set of cities.

[Table 30-1](#) shows the valid map layers for each base map.

Table 30-1 Prebuilt Base Maps and Layers

Base Map	Layers
usa	country, states, counties
world	continents, countries
worldRegions	regions, countries
africa, asia, australia, europe, northAmerica, southAmerica	continents, countries
apac, emea, latinAmerica, usaAndCanada	regions, countries, cities

When you are binding your data collection to a thematic map, you must provide a column in the data model that specifies the location of the area or point data using the map location Ids of the regions from the base map for which the data is being displayed. Area locations are specified in the `name` attribute of the `areaLocation` component, and point locations are specified in the `pointName` attribute for the `pointLocation` component when its `type` attribute is set to `pointName`.

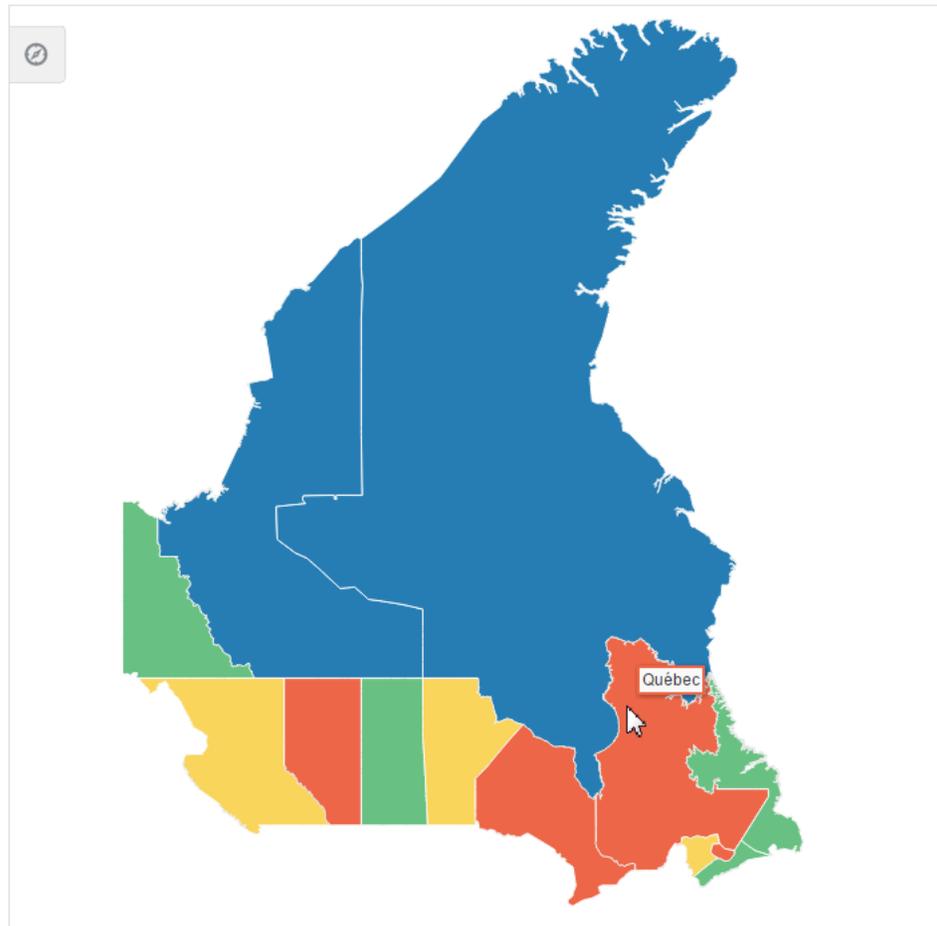
You can download a comma-separated value (CSV) file for each of the prebuilt map layers with a complete listing of all the thematic map base map location Ids. Find these links in the tag documentation for the `areaLocation` component, `name` attribute. To access tag documentation for the data visualization components, select the component in the Structure window and click the help button in the Properties window.

For more information, see the "What You May Need to Know About Base Map Location Ids" section in the *Developing Fusion Web Applications with Oracle Application Development Framework*.

Defining a Custom Base Map Using Map Provider APIs

In addition to using the prebuilt base maps, thematic maps can be configured to retrieve geographic data from any data source including a database or an `eLocation` service such as Oracle MapViewer or geocoder service using any file type. The thematic map component `mapProvider` attribute requires an object of type `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.MapProvider`. The `MapProvider` APIs allow the custom base map to be configured and used like a prebuilt base map with the same functionality including drilling, labels, custom region support, and point layers.

For example, [Figure 30-26](#) displays a custom Canada base map that uses the `mapProvider` attribute to retrieve geographic data from a GeoJSON formatted zip file.

Figure 30-26 Thematic Map Custom Base Map

Use these abstract and utility classes to configure a custom basemap:

- `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.MapProvider`: Abstract class that provides the APIs to render a custom basemap. Use an EL Java callback to pass an implementation of this class to the `mapProvider` attribute on `dvt:thematicMap`.

To view an implementation of the `mapProvider` class for the Canada custom base map example, see [Sample Code for Thematic Map Custom Base Map](#).

- `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.utils.MapProviderUtils`: Utility class that provides APIs for converting Java 2D objects to a SVG path command.
- `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.LayerArea`: Abstract class that provides APIs to get the data that the thematic map component requires to render a `dvt:areaLayer` and is used by the `getLayerAreas()` and `getChildAreas()` methods on the abstract `MapProvider` class.

To view an implementation of the `layerArea` class for the Canada custom base map example, see [Sample Code for Thematic Map Custom Base Map Area Layer](#).

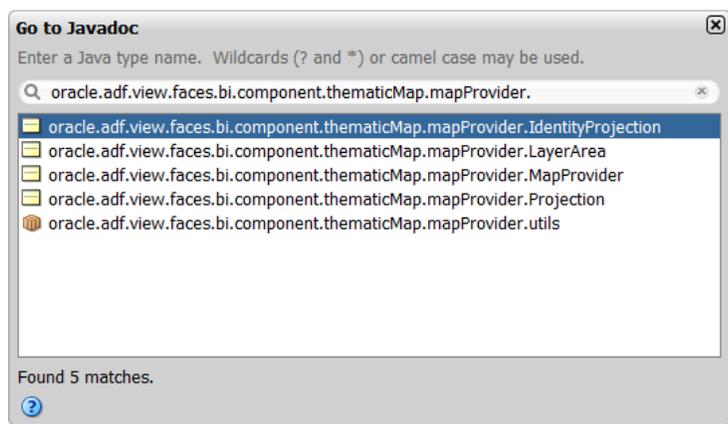
- `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.Projection`: If required, abstract class that provides an API for projecting points from the

`dvt:pointDataLayer` for a custom base map and is used by the `getProjection()` method on the abstract `MapProvider` class. An identity projection class (`oracle.adf.view.faces.bi.component.thematicMap.mapProvider.IdentityProjection`) will be provided and returned by default by the `MapProvider` class if no point projections need to be performed.

You can view the complete Javadoc for these classes in JDeveloper. From the toolbar select **Navigate > Go to Javadoc** and enter the class name in the search field.

Figure 30-27 shows the search dialog.

Figure 30-27 Custom Base Map APIs in Javadoc



On the JSF page, the code for the thematic map using the Canada custom base map is as follows:

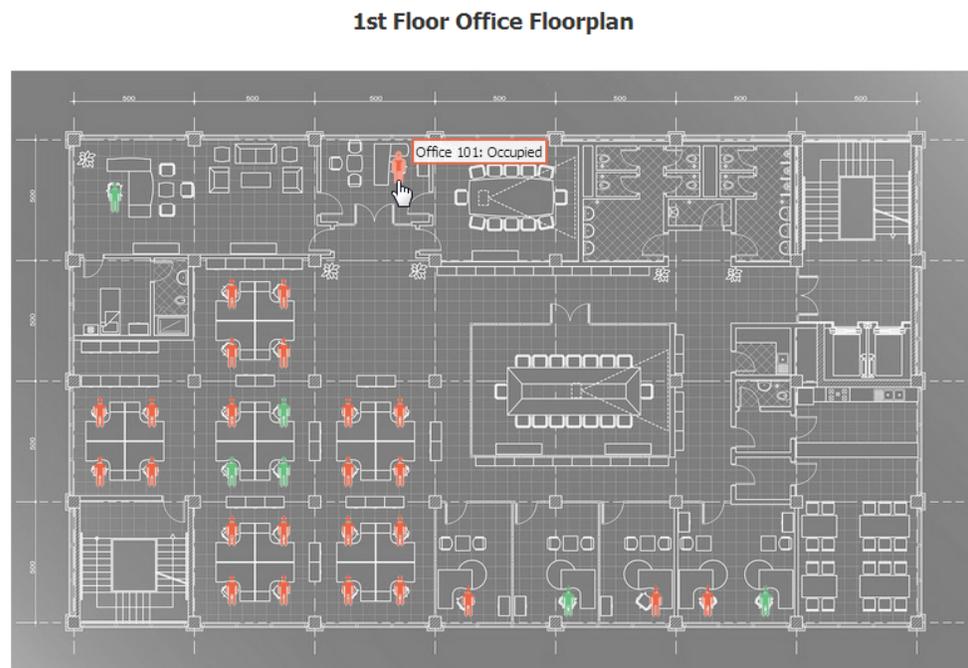
```
<dvt:thematicMap id="tm1" summary="Map Provider" basemap="canada"
    mapProvider="#{mapProviderBean.callback}"
initialZooming="auto">
  <af:transition transition="auto" triggerType="display"/>
  <dvt:areaLayer layer="territories" id="all" labelDisplay="off">
    <dvt:areaDataLayer selectionMode="none" id="adl1"
      value="#{mapProviderBean.territories}" var="row"
      varStatus="rowStatus">
      <dvt:areaLocation id="l1" name="#{row.id}">
        <dvt:area id="a1" value="#{row.value}">
          <dvt:attributeGroups id='agl' type="color"
            value="#{row.categories[0]}"
            label="#{row.categories[0] == 'Group 1' ?
              'Territory C' :
              (row.categories[0] == 'Group 2' ?
                'Territory A' : 'Territory B')}" />
          </dvt:area>
        </dvt:areaLocation>
      </dvt:areaDataLayer>
    </dvt:areaLayer>
  </dvt:thematicMap>
```

Defining a Custom Base Map Using Image Files

The thematic map component also supports the definition of a custom base map using an XML metadata file to define data points on an associated image.

For example, the thematic map in [Figure 30-28](#) shows a custom base map of an office floor plan with points defined for each office location. In the example, a tooltip identifies if the office space is occupied or available.

Figure 30-28 Custom Base Map with Defined Point Layer



To define a custom base map using an image file, you must create an xml metadata file that defines the image to use for the map and add the file to your project source. In the office floor plan example, a static points for each office is specified by number, as shown in the example below.

```
<basemap id='floorplan'>
  <layer id='floor1'>
    <image height='813' width='1300'
      source='/resources/images/thematicMap/floorplan.jpg' />
  </layer>
  <points>
    <point x='140' y='170' name='100' longLabel='Office 100' />
    <point x='523' y='130' name='101' longLabel='Office 101' />
    <point x='298' y='300' name='102' longLabel='Office 102' />
    <point x='368' y='300' name='103' longLabel='Office 103' />
    <point x='298' y='380' name='104' longLabel='Office 104' />
    <point x='368' y='380' name='105' longLabel='Office 105' />
    <point x='120' y='460' name='106' longLabel='Office 106' />
    <point x='190' y='460' name='107' longLabel='Office 107' />
  </points>
</basemap>
```

```

<point x='120' y='540' name='108' longLabel='Office 108' />
<point x='190' y='540' name='109' longLabel='Office 109' />
<point x='298' y='460' name='110' longLabel='Office 110' />
<point x='368' y='460' name='111' longLabel='Office 111' />
<point x='298' y='540' name='112' longLabel='Office 112' />
<point x='368' y='540' name='113' longLabel='Office 113' />
<point x='455' y='460' name='114' longLabel='Office 114' />
<point x='525' y='460' name='115' longLabel='Office 115' />
<point x='455' y='540' name='116' longLabel='Office 116' />
<point x='525' y='540' name='117' longLabel='Office 117' />
<point x='298' y='620' name='118' longLabel='Office 118' />
<point x='368' y='620' name='119' longLabel='Office 119' />
<point x='298' y='700' name='120' longLabel='Office 120' />
<point x='368' y='700' name='121' longLabel='Office 121' />
<point x='455' y='620' name='122' longLabel='Office 122' />
<point x='525' y='620' name='123' longLabel='Office 123' />
<point x='455' y='700' name='124' longLabel='Office 124' />
<point x='525' y='700' name='125' longLabel='Office 125' />
<point x='617' y='715' name='126' longLabel='Office 126' />
<point x='752' y='715' name='127' longLabel='Office 127' />
<point x='870' y='715' name='128' longLabel='Office 128' />
<point x='940' y='715' name='129' longLabel='Office 129' />
<point x='1018' y='715' name='130' longLabel='Office 130' />
</points>
</basemap>

```

In the thematic map component you then specify an area layer which points to the definition in the metadata file using the `basemap` attribute. You can then define an area layer with a specify an area layer with named points, as shown in the example below.

```

<dvt:thematicMap id="tml" summary="Custom Base Map" basemap="floorplan"
    animationOnDisplay="none"
    source="/resources/images/thematicMap/offices.xml"
    inlineStyle="background-color:transparent;height:540px;
                width:810px;"
    controlPanelBehavior="hidden" panning="none"
zooming="none">
    <dvt:areaLayer layer="floor1" id="all">
        <dvt:pointDataLayer id="pd11" selectionMode="single"
partialTriggers=":::t1"
                value="{officesBean.currentFloor.offices}"
var="row"
                animationOnDataChange="alphaFade"
varStatus="rowStatus">
            <dvt:pointLocation id="pl1" pointName="{row.id}" type="pointName">
                <dvt:marker id="m1" shape="human" opacity="1" gradientEffect="none"
                    fillColor="{row.color}" scaleX="3" scaleY="3"
                    shortDesc="{row.categories[0]}" />
            </dvt:pointLocation>
        </dvt:pointDataLayer>
    </dvt:areaLayer>
</dvt:thematicMap>

```

In the metadata file you can also specify different images for different screen resolutions and display directions. The thematic map component chooses the correct image for the layer based on the screen resolution and direction. The display direction can be either left-to-right or right-to-left. The default direction for the image is left-to-right, which you can change to right-to-left by setting the `dir` attribute of the `image` component to `rtl`. The example below shows sample code for a metadata file with images for a custom base map at different screen resolutions.

```
<basemap id="car" >
  <layer id="exterior" >
    <image source="/maps/car-800x800.png"
      width="2560"
      height="1920" />
    <image source="/maps/car-800x800-rtl.png"
      width="2560"
      height="1920"
      dir="rtl" />
    <image source="/maps/car-200x200.png"
      width="640"
      height="480" />
    <image source="/maps/car-200x200-rtl.png"
      width="640"
      height="480"
      dir="rtl" />
  </layer>
</basemap>
```

You can define an area layer with a point data layer, or define the point data layer as a direct child of the thematic map component. If the point layer is nested inside a map layer, the data points are only displayed when that map layer is displayed. When a point data layer is configured as a direct child of the thematic map component, the data points are always displayed as a global point layer. The example below shows sample code for a thematic map with a point data layer defined for an area layer.

```
<dvt:thematicMap id="tml" basemap="car" source="customBasemaps/map1.xml" >
  <dvt:areaLayer id="all" layer="exterior" >
    <dvt:pointDataLayer id="pdl1"
      var="row"
      value="{bindings.thematicMapData.collectionModel}" >
      <dvt:pointLocation id="pl1"
        type="pointXY"
        pointX="{row.x}"
        pointY="{row.y}" >
        <dvt:marker id="m1" fillColor="#FFFFFF" shape="circle" />
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

The example below shows sample code for a thematic map with a point data layer as a direct child of the thematic map component.

```
<dvt:thematicMap id="demo1" basemap="car" source="customBasemaps/map1.xml"
>
  <dvt:areaLayer id="all" layer="exterior" />
  <dvt:pointDataLayer id="pd1"
    var="row"
    value="{bindings.thematicMapData.collectionModel}" >
    <dvt:pointLocation id="pl1"
      type="pointXY"
      pointX="{row.x}"
      pointY="{row.y}" >
      <dvt:marker id="m1" fillColor="#FFFFFF" shape="circle" />
    </dvt:pointLocation>
  </dvt:pointDataLayer>
</dvt:thematicMap>
```

Customizing Thematic Map Display Attributes

You can customize the display attributes of ADF DVT thematic maps labels, including prebuilt map layer labels and labels for area and marker components that stamp out the data values for the thematic map. You can also configure tooltips to display data when the user moves the cursor over the map.

Data values that require special formatting, for example, currency or percentages, can be configured to display special symbols and decimal points.

How to Customize Thematic Map Labels

By default, each region in each map layer for a prebuilt base map has a label with a short and long type, for example, BRA and Brazil in the `countries` layer of the world base map.

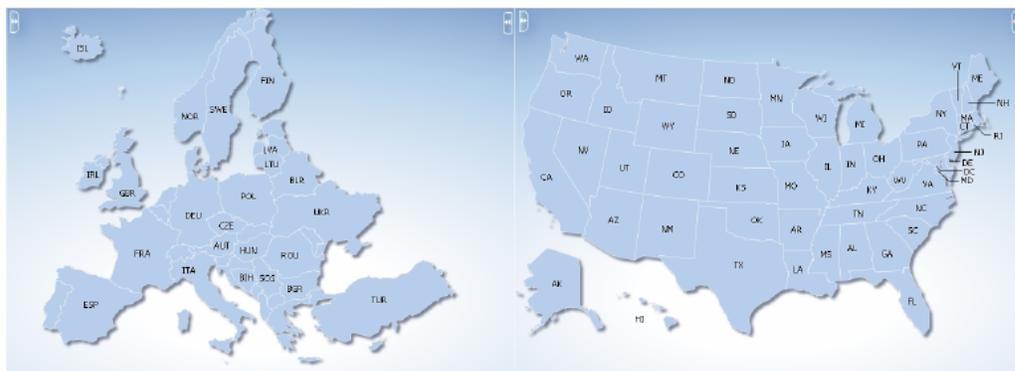
The map layer is specified by the `areaLayer` component in the `layer` attribute, for example:

```
<dvt:areaLayer id="all" layer="countries">
```

Note:

Labels are only displayed when they fit within the named region of the map layer. When a label does not fit inside the region, it is not displayed unless leader lines determining the alternate label location are provided for that layer in the base map. Only the prebuilt `usa` map provides leader lines for labels outside the `states` layer.

Figure 30-29 shows the default labels for the `europa` and `usa` base maps.

Figure 30-29 Default Labels for Europe and USA Base Maps

You can customize the default labels provided by the base map for display and style.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

To customize a map layer label:

1. In the Structure window, right-click the **dvt:areaLayer** component representing the map layer for which you wish to customize the label, and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section, and set the following attributes:
 - **LabelDisplay**: Use the dropdown list to select how the prebuilt base map labels for the layer are to be displayed. Valid values are: **auto** (default) to display the label if there is sufficient space in the region; **on** to display the base map labels for all regions of this layer; and **off** to disable the display of labels.
 - **LabelStyle**: Enter the font-related CSS styles to use for the label font.
 - **LabelType**: Use the dropdown list to select the prebuilt base map labels to display. Valid values are **short** (default) to display the short labels defined in the base map, for example, **TX**, and **long** to display the long labels defined in the base map, for example, *Texas*.

You can also override the default labels in the base map by specifying attributes for the `area` components that stamp out stylistic attributes such as fill colors, patterns, or opacity onto the geographic regions of the map, and for `marker` components that stamp out built-in or custom shapes associated with data points on the map.

To customize area and marker labels:

1. In the Structure window, right-click the **dvt:area** component or **dvt:marker** component representing the stamp for which you wish to customize the label, and choose **Go to Properties**.

2. In the Properties window, expand the **Other** section, and set the following attributes:
 - **LabelDisplay**: Use the dropdown list to select **on** to display the text displayed in the **value** attribute.
 - **Value**: Enter the text you wish to use for the area or marker label when **LabelDisplay** is set to **on**.
 - **LabelStyle**: Enter the font-related CSS styles to use for the area or marker label font.
 - **LabelPosition**: Available only for the marker label. Use the dropdown list to select the position relative to the marker that the specified value label should be displayed. Valid values are **center** (default), **top**, and **bottom**.

 **Note:**

If a marker is displayed on a region, that is as a child component to a `pointLocation` for an `areaDataLayer`, and that marker has a label, then the label associated with the base map region will not be displayed.

- **ShortDesc**: Enter the short description you wish to use for the area or marker stamp. This value is used for the tooltip that displays when the user moves the cursor over the area or marker. For more information, see [How to Configure Tooltips to Display Data](#).

How to Configure Tooltips to Display Data

By default, thematic maps automatically display tooltips using map layer labels when the user moves the cursor over the map. If data is available, the map layer label is concatenated with the value from the `area` or `marker` component stamp. You can also configure the tooltip to only display available data.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

To configure tooltips to display data:

1. In the Structure window, right-click the **dvt:thematicMap** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. For the `TooltipDisplay` attribute choose **auto** to display the label concatenated with the value of the area or marker stamp, or **shortDesc** to only display the value.
3. In the Structure window, right-click the **dvt:area** component or **dvt:marker** component representing the stamp for which you wish to display data in the tooltip, and choose **Go to Properties**.

4. In the Properties window, expand the **Other** section. For the **ShortDesc** attribute enter the value you want to display in the tooltip. For example, if an area component `value` attribute is `#{row.data}`, use that same value for the `shortDesc` attribute.

How to Format Numeric Data Values in Area and Marker Labels

Thematic map `area` and `marker` components can display numeric data values in labels, for example a dollar value, or a percentage. Area and marker labels are specified in the `value` attribute of the component.

You can format numeric data values by adding a standard ADF converter, `af:convertNumber`, as a child of the area or marker component, or by specifying a converter through an EL Expression directly on the component. If both a converter and a child `af:convertNumber` tag are specified, then the properties of the child tag take precedence.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

You should already have configured an area or marker label in your thematic map. If you do not, follow the instructions in this chapter to customize an area or marker label. For more information, see [How to Customize Thematic Map Labels](#).

To format numeric data values in area or maker labels:

1. In the Structure window, right-click the **dvt:area** or **dvt:marker** component representing the stamp you wish to format, and choose **Insert Inside Area** or **Insert Inside Marker** > **Convert Number**.
2. In the Properties window, specify values for the attributes of the **af:convertNumber** component to produce numeric formatting. Use the help button to display the complete tag documentation for the `af:convertNumber` component.

The example below shows sample code for formatting numeric data values for an area and a marker label.

```
...
<dvt:area id="a2" labelDisplay="on" value="#{mapBean.value}" >
  <af:convertNumber id="cn1" type="currency"/>
</dvt:area>
<dvt:marker id="m2" labelDisplay="on" value="#{mapBean.value}" >
  <af:convertNumber id="cn1" type="currency"/>
</dvt:marker>
...
```

Alternatively, specify a converter through an EL expression directly on the area or marker component. For example:

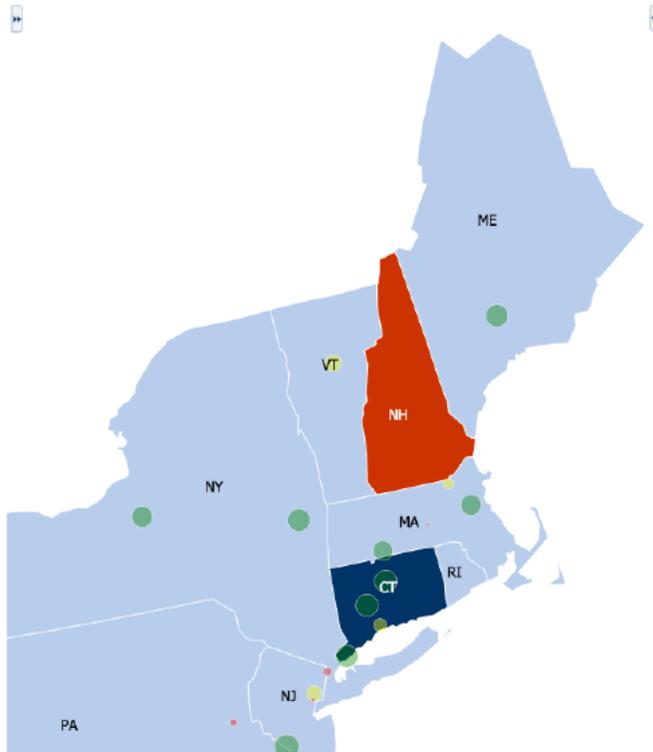
```
<dvt:marker id="m1" labelDisplay="on" value="#{mapBean.value}"  
  converter="#{mapBean.myConverter}"/>
```

How to Configure Thematic Map Data Zooming

Thematic maps support customized data zooming features. You can configure the initial zoom of a thematic map to fit the rendered area and point data layers without displaying the entire base map. You can also configure a thematic map to render and zoom on an isolated data area.

For example, the thematic map in [Figure 30-30](#) displays area and point data layers for US states. In the example, data is not rendered for all states in the US base map. The initial zoom fits to the boundaries of the data layers.

Figure 30-30 Thematic Map with Initial Zoom on Data Layers



To set the initial zoom to fit rendered data layers, on the `dvt:thematicMap` component, set the `initialZooming` property to `auto`.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map with a defined data layer on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

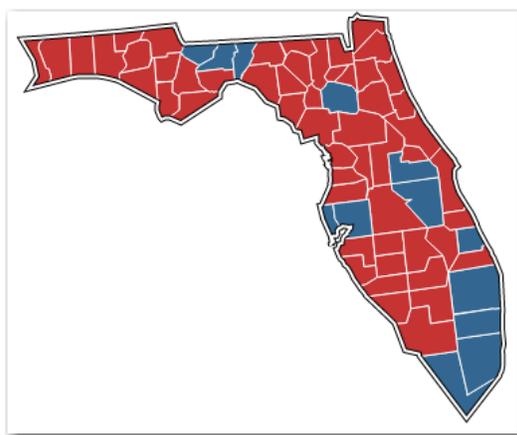
To configure the initial zoom:

1. In the Structure window, right-click the `dvt:thematicMap` component, and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section, and set the **InitialZooming** property to `auto` to automatically focus the zoom on the rendered data layers. The default value is `none`.

Use the help button to display the complete tag documentation for the `dvt:thematicMap` component.

The thematic map in [Figure 30-31](#) shows the 2012 presidential election results by county for the state of Florida. In the example, an `isolatedRowKey` property on the `dvt:areaDataLayer` is set to isolate the state of Florida on the states area layer and US base map. The `disclosedRowKey` attribute on the `dvt:areaDataLayer` is set to show Florida drilled down to the counties area layer and stylized with color to represent the winning candidate's color for each county.

Figure 30-31 Thematic Map of 2012 Presidential Election Results for Florida



To configure zoom on an isolated data area:

1. In the Structure window, right-click the `dvt:areaDataLayer` component, and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section, and set the **IsolatedRowKey** property to the area corresponding to the isolated row key for which you wish to display data.
3. For the **DisclosedRowKey** property, specify the row keys for the disclosed regions of the data layer. Each entry in the set is a row key.

The example below shows the code for configuring the thematic map in [Figure 30-31](#).

```
<dvt:thematicMap id="thematicMap" basemap="usa" drilling="off"
  animationOnDisplay="alphaFade"
  controlPanelBehavior="hidden" summary="presidential map"
  inlineStyle="background-color:transparent;
  width:400px;border:none;">
  <dvt:areaLayer id="all" layer="states">
    <dvt:areaDataLayer id="ad11"
      disclosedRowKeys="#{electionBean.disclosedRowKey}"
      contentDelivery="lazy"
```

```

        isolatedRowKey="#{electionBean.isolatedRowKey}"
        value="#{electionBean.state}" var="row"
        varStatus="rowStatus">
    <dvt:areaLocation id="loc1" name="#{row.id}">
        <dvt:area id="a1" fillColor="#{row.color}"/>
    </dvt:areaLocation>
</dvt:areaDataLayer>
</dvt:areaLayer>
<dvt:areaLayer id="a12" layer="counties">
    <dvt:areaDataLayer id="ad12" value="#{electionBean.counties}"
var="row"
        varStatus="rowStatus"
        selectionMode="single" contentDelivery="lazy">
    <dvt:areaLocation id="loc2" name="#{row.id}">
        <dvt:area id="a2" fillColor="#{row.color}"/>
    </dvt:area>
    </dvt:areaLocation>
</dvt:areaDataLayer>
</dvt:areaLayer>
</dvt:thematicMap>

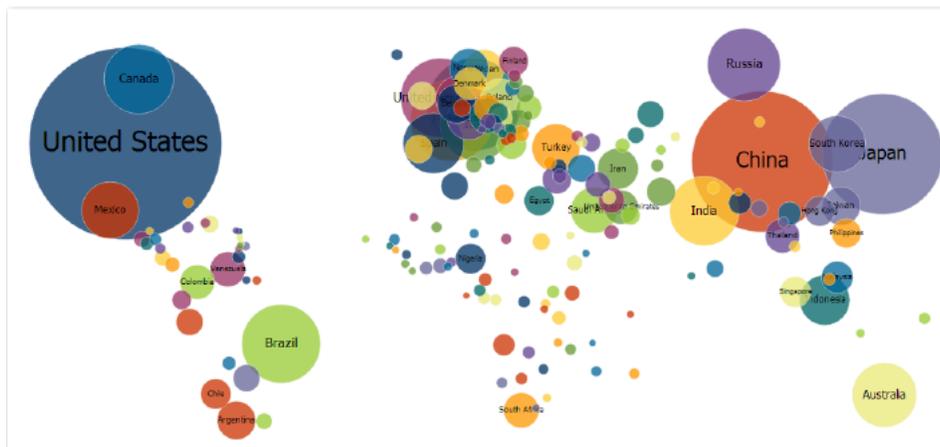
```

How to Configure Invisible Area Layers

By design, thematic maps focus on data within a defined geographic location. In some instances you may wish to display your data without showing the default background color or boundaries of the area layer.

For example, the thematic map in [Figure 30-32](#) displays global GDP (gross domestic product) per capita by country in millions of US dollars using markers scaled by size and label font. In the example, displaying background colors or boundaries of the countries represented by markers do not contribute to an understanding of the data and may in fact, distract from the overall display.

Figure 30-32 Thematic Map Displaying Global GDP Per Capita



You can override the default area layer color and border display without using a skinning key by setting the `areaStyle` property CSS attributes for `background-color` and `border-color` to transparent on the `dvt:areaLayer` component.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map with a defined area layer on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

To configure invisible area layers:

1. In the Structure window, right-click the **dvt:areaLayer** component representing the geographic area you wish to you wish to format, and choose **Go to Properties**.
2. In the Properties window, expand the **Other** section, and for the **AreaStyle** property, set semi-colon separated CSS values `background-color` and `border-color` to `transparent`. For example:

```
background-color:transparent;border-color:transparent
```

Use the help button to display the complete tag documentation for the `dvt:areaLayer` component.

The thematic map in [Figure 30-32](#) uses scaled markers and associated labels to represent a country's GDP per capita. In the example, a minimum GDP value is set to not show labels on smaller markers. The example below shows the code for configuring the markers and labels.

```
<dvt:marker id="m1" scaleX="#{(5 + row.categories[0]/25)}"
  value="#{row.categories[2]}"
  labelStyle="font-size:#{row.categories[0]/25 + 30}px;"
  opacity="0.75"
  labelDisplay="#{row.categories[0]/25 le 20 ? 'off' : 'on'}"
  gradientEffect="none"
  scaleY="#{(5 + row.categories[0]/25)}"
  shortDesc="\${row.categories[1]} Million">
  <dvt:attributeGroups id="ag1" type="color" value="#{row.id}"/>
</dvt:marker>
```

Note:

To maintain the relative size of the scaled markers and labels to the thematic map when it is zoomed in or out, set the `dvt:thematicMap` component `markerZoomBehavior` property to `zoom`.

For detailed information about configuring markers to represent thematic map data, see the "Styling Areas, Markers, and Images to Display Data" section of *Developing Fusion Web Applications with Oracle Application Development Framework*.

What You May Need to Know About Skinning and Customizing the Appearance of a Thematic Map

Thematic maps also support skinning to customize the color and font styles for thematic map layers, areas, and markers. In addition, you can use skinning to define the styles for a thematic map area when the user hovers the mouse over or selects it.

The example below shows the skinning key for a thematic map area configured to show the border color in red when the user selects it.

```
af|dvt-area:selected
{
  -tr-border-color:#0000FF;
}
```

For the complete list of thematic map skinning keys, see the *Oracle Fusion Middleware Documentation Tag Reference for Oracle Data Visualization Tools Skin Selectors*. To access the list from JDeveloper, from the Help Center, choose Documentation Library, and then Fusion Middleware Reference and APIs. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactive Features to Thematic Maps

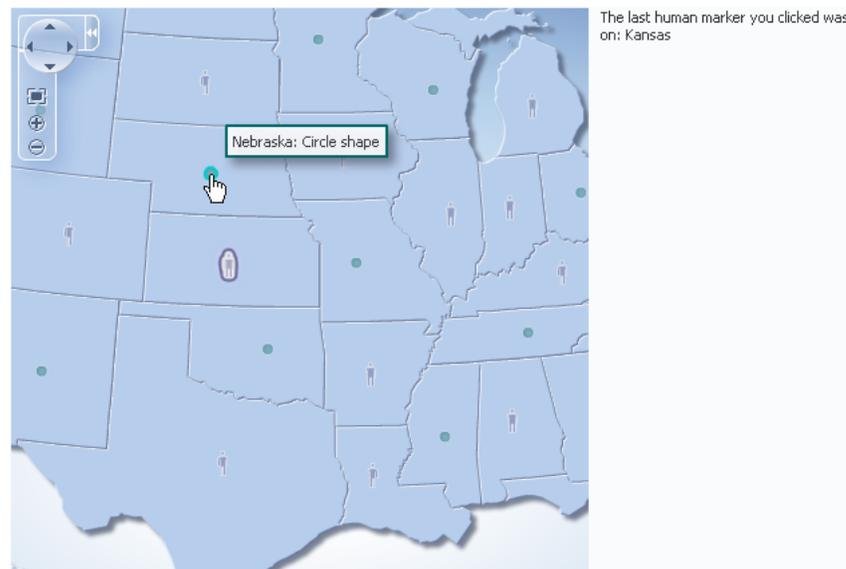
ADF DVT Thematic maps include support for interaction features including selection and action events, drilling, popups, animation, and drag and drop operations.

How to Configure Selection and Action Events in Thematic Maps

You can configure your thematic map components to allow users to select one or more areas or markers across multiple data layers. By default, selection is not enabled. You configure selection on `areaDataLayer` and `pointDataLayer` components to allow selection of one or multiple area or marker stamps.

Once selection is enabled, you can configure an area or marker stamp with an action listener to specify and handle a custom event such as displaying output text or navigating to another page. For more information about ADF action events, see [Handling Events](#).

[Figure 30-33](#) shows a thematic map configured to display output text when a human marker is clicked, and navigate to another JSF page when a circle marker is clicked.

Figure 30-33 Thematic Map Action Events

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

You should have already configured a data layer with an area or marker to display data on your thematic map.

To configure selection and action events:

1. In the Structure window, right-click the **dvt:areaDataLayer** or **dvt:pointDataLayer** component and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section. For the **SelectionMode** attribute choose **single** to enable single selection of an area or marker, or **multiple** to enable multiple selection of areas or markers.
3. In the Structure window, right-click the **dvt:area component** or **dvt:marker** component representing the stamp for which you wish to configure an action event, and choose **Go to Properties**.
4. In the Properties window, expand the **Behavior** section. In this section set the following attributes:
 - **Action:** Enter a reference to an action method sent by the component, or the static outcome of an action. For example, `mapAction`.
 - **ActionListener:** Enter a method reference to an action listener. For example, `#{tmapEventBean.processClick}`

The example below shows sample code for configuring markers to fire action events.

```
<f:facet name="center">
  <dvt:thematicMap id="thematicMap"
    imageFormat="flash" basemap="usa"
    inlineStyle="width:98%;height:95%;"
    summary="Thematic map showing action events">
    <dvt:areaLayer id="areaLayer" layer="states"
      labelDisplay="off">
      <dvt:areaDataLayer id="dataLayer"
        contentDelivery="immediate"
        value="#{tmapBean.colorModel}"
        var="row"
        varStatus="rowStatus"
        selectionMode="single">
        <dvt:areaLocation id="dataLoc"
          name="#{row.name}">
          <dvt:marker id="marker1"
            shape="human" scaleX="3"
            scaleY="3"
            fillColor="#666699"
            actionListener="#{tmapEventBean.processClick}"
            rendered="#{row.category == 'category1'}"
            shortDesc="Human shape"/>
          <dvt:marker id="marker2"
            shape="circle"
            scaleX="2" scaleY="2"
            fillColor="#006666"
            action="mapAction"
            rendered="#{row.category == 'category2'}"
            shortDesc="Circle shape"/>
          </dvt:areaLocation>
        </dvt:areaDataLayer>
      </dvt:areaLayer>
    </dvt:thematicMap>
  </f:facet>
</f:facet name="end">
  <af:outputText value="#{tmapEventBean.clickString}"
    id="ot1"

partialTriggers="thematicMap:areaLayer:dataLayer:marker1"/>
</f:facet>
```

You can also configure an area or point data layer with a selection listener for declarative master-detail processing, for example, to display the thematic map associated data in another UI component on the page such as a table. For more information, see the "What You May Need to Know About Configuring Master-Detail Relationships" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

How to Add Popups to Thematic Map Areas and Markers

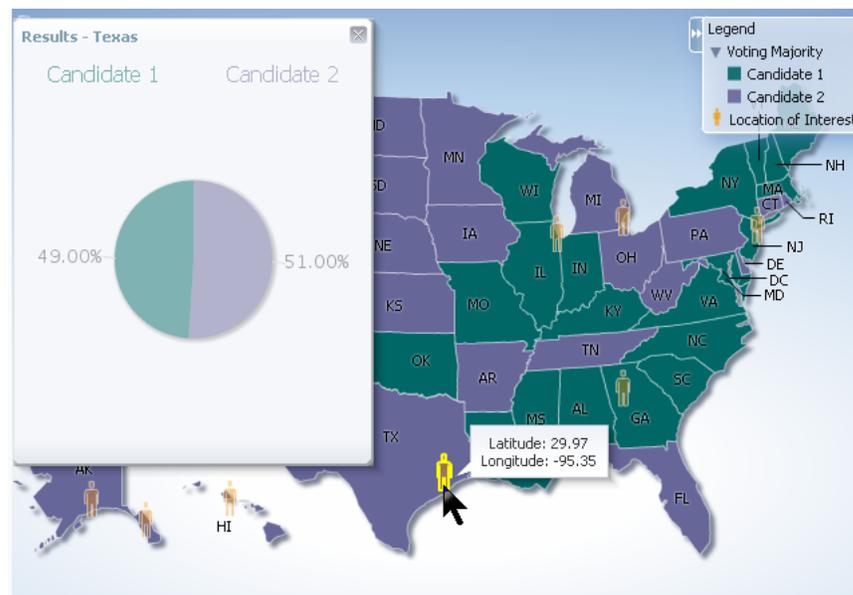
Thematic map `area` and `marker` components can be configured to display popup dialogs, windows, and menus that provide information or request input from end users.

Using the `af:popup` component with other ADF Faces components, you can configure functionality to allow your end users to show and hide information in secondary windows, input additional data, or invoke functionality such as a context menu.

With ADF Faces components, JavaScript is not needed to show or hide popups. The `af:showPopupBehavior` tag provides a declarative solution, so that you do not have to write JavaScript to open a `popup` component or register a script with the `popup` component. For more information about these components, see [Using Popup Dialogs, Menus, and Windows](#).

For example, you may want to associate a popup to display information in a dialog or note window with thematic map areas or markers. [Figure 30-34](#) shows a thematic map area (Texas) clicked to display a dialog of data about voting results, and the cursor hovered over marker (human) displaying a note window of data about a specific location.

Figure 30-34 Area Dialog and Marker Note Window



Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

You should already have created the popup components for the thematic map area or marker components to reference. The example below shows sample code for the dialog to be referenced when an area stamp is clicked.

```
<af:popup id="pop1" contentDelivery="lazyUncached" launcherVar="source"
  eventContext="launcher">
  <af:setPropertyListener from="#{tmapPopupBean.colorModel.rowData}"
```

```

        to="{tmapPopupBean.source}"
        type="popupFetch"/>
<af:dialog id="nw1" modal="false" type="none"
    title="Results - #{tmapPopupBean.source.fullName}">
<af:panelGroupLayout id="pg16">
<af:panelGroupLayout id="pg17" layout="horizontal"
    halign="center">
    <af:outputText value="Candidate 1" id="ot2"
        inlineStyle="color:#{tmapPopupBean.strColor2};
        font-size:medium;"/>
    <af:spacer width="50" height="10" id="spacer1"/>
    <af:outputText value="Candidate 2" id="ot1"
        inlineStyle="color:#{tmapPopupBean.strColor1};
        font-size:medium;"/>
</af:panelGroupLayout>
<af:panelGroupLayout id="pg15" layout="horizontal"
    halign="center">
    <dvt:pieGraph id="graph1" subType="PIE"
        inlineStyle="height:250.0px;width:250.0px"

tabularData="{tmapPopupBean.graphData[tmapPopupBean.source]}"
        imageFormat="PNG">
    <dvt:background fillTransparent="true"/>
    <dvt:graphPieFrame fillTransparent="true"/>
    <dvt:seriesSet>
        <dvt:series index="0" color="{tmapPopupBean.color1}"/>
        <dvt:series index="1" color="{tmapPopupBean.color2}"/>
    </dvt:seriesSet>
    <dvt:sliceLabel rendered="true">
    <dvt:graphFont id="graphFont1" size="14"/>
    </dvt:sliceLabel>
    <dvt:pieLabel rendered="false"/>
    <dvt:legendArea rendered="false"/>
    </dvt:pieGraph>
</af:panelGroupLayout>
</af:panelGroupLayout>
</af:dialog>
</af:popup>

```

The example below shows sample code for the note window to be referenced when the user hovers the mouse over a marker stamp.

```

<af:popup id="pop2" contentDelivery="lazyUncached" launcherVar="source"
    eventContext="launcher">
    <af:setPropertyListener from="{tmapPopupBean.pointModel.rowData}"
        to="{tmapPopupBean.noteSource}"
        type="popupFetch"/>
    <af:noteWindow id="nw2">
    <af:panelGroupLayout id="pg18" halign="center" layout="vertical">
    <af:outputText value="Latitude: #{tmapPopupBean.noteSource.latitude}"
        id="ot4"/>
    <af:outputText value="Longitude:
#{tmapPopupBean.noteSource.longitude}"
        id="ot5"/>

```

```

    </af:panelGroupLayout>
  </af:noteWindow>
</af:popup>

```

For more information about popup components, see [Using Popup Dialogs, Menus, and Windows](#).

To add a popup to an area or marker:

1. In the Structure window, right-click the **dvt:area** or **dvt:marker** component and choose **insert Inside Area** or **Insert Inside Marker > Show Popup Behavior**.
2. In the Properties window, set the following attributes:
 - **PopupId**: Enter the ID of the popup referenced by the area or marker component. An ID beginning with a colon will be treated as absolute after trimming off the colon.
 - **TriggerType**: Enter the event type that will trigger the popup being displayed. Valid values for thematic map area or marker components are **action**, **click** and **mouseHover**.
 - **Align**: From the dropdown list, choose how the popup should be aligned with the area or marker component.
 - **AlignID**: Enter the ID of the area or marker component associated with the popup. An ID beginning with a colon will be treated as absolute after trimming off the colon.

The example below shows sample code for adding popup components to the area and marker stamps in a thematic map.

```

<dvt:thematicMap id="thematicMap" imageFormat="flash
    basemap="usa" summary="Thematic map showing voting data
in US">
  <dvt:legend label="Legend">
    <dvt:showLegendGroup label="Voting Majority">
      <dvt:legendSection source="areaLayer:dataLayer:areal"/>
    </dvt:showLegendGroup>
    <dvt:legendSection source="areaLayer:pointLayer:marker1"/>
  </dvt:legend>
  <dvt:areaLayer id="areaLayer" layer="states">
    <dvt:areaDataLayer id="dataLayer" contentDelivery="immediate"
      value="{tmapPopupBean.colorModel}"
      var="row" varStatus="rowStatus">
      <dvt:areaLocation id="areaLoc" name="{row.name}">
        <dvt:area id="areal"
          fillColor="{row.value > 50 ? tmapPopupBean.color1 :
            tmapPopupBean.color2}"
          <f:attribute name="legendLabel" value="{row.value > 50 ?
'Candidate 2' :
          'Candidate 1'}" />
          <af:showPopupBehavior triggerType="click"
            popupId=":::pop1"
            alignId="areal"
            align="endAfter"/>
        </dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>

```

```

</dvt:areaDataLayer>
<dvt:pointDataLayer id="pointLayer"
    value="#{tmapPopupBean.pointModel}" var="row"
    varStatus="rowStatus"
    contentDelivery="immediate">
    <dvt:pointLocation id="pointLoc" type="pointXY"
        pointX="#{row.longitude}"
        pointY="#{row.latitude}">
        <dvt:marker id="marker1" shape="human" fillColor="#FF9900"
            scaleX="3" scaleY="3"
            <f:attribute name="legendLabel" value="Location of Interest" />
            <af:showPopupBehavior
                triggerType="mouseHover"
                alignId="marker1"
                popupId=":::pop2"
                align="endAfter" />
        </dvt:marker>
    </dvt:pointLocation>
</dvt:pointDataLayer>
</dvt:areaLayer>
</dvt:thematicMap>

```

How to Configure Animation Effects

By default, thematic maps are animated upon initial rendering of the map, when the data associated with the map changes, and when a region is drilled in the map. You can customize the default setting of each animation event.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

To customize animation effects in a thematic map:

1. In the Structure window, right-click the **thematicMap** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. Use this section to set the following attributes:
 - **AnimationDuration**: Enter the animation duration in milliseconds. The default value is **1000**.
 - **AnimationOnDisplay**: Use the dropdown list to select the animation effect upon initial display of the thematic map. The default value is **zoom**.
 - **AnimationOnDrill**: Use the dropdown list to select the animation effect when a map layer is drilled to a lower level. The default value is **alphaFade**.
 - **AnimationOnMapChange**: Use the dropdown list to select the animation effect when the value of the area or point data layer changes, or when the base map changes. The default value is **none**.

Table 30-2 shows the animation effect available for each supported thematic map event.

Table 30-2 Thematic Map Animation Effects

Animation Effect	AnimationOnDisplay	AnimationOnDrill	AnimationOnMapChange
none	x	x	x
alphaFade	x	x	x
conveyorFromLeft	x		x
conveyorFromRight	x		x
cubeToLeft	x		x
cubeToRight	x		x
flipLeft	x		x
flipRight	x		x
slideToLeft	x		x
slideToRight	x		x
transitionToLeft	x		x
transitionToRight	x		x
zoom	x		x

How to Add Drag and Drop to Thematic Map Components

The ADF Faces framework provides the ability to drag and drop items from one place to another on a page. For thematic maps, `area` and `marker` components can be used as a drag source by adding and configuring a child `af:dragSource` component, and `areaLayer` components can be used as a drop target by adding and configuring a child `af:dropTarget` component.

For example, you could drag an area representing the population for a state in a USA map and drop it into a table to display the data.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child components can affect functionality. For more information, see [Configuring Thematic Maps](#).

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see [How to Add a Thematic Map to a Page](#).

To use an area or marker as a drag source:

1. In the Structure window, right-click the `area` or `marker` component you are configuring as a drag source, and choose **Insert Inside Area** or **Insert Inside Marker > Drag source**.
2. In the Properties window, specify the **actions** attribute.

The example below shows sample code for adding and configuring an area as a drag source.

```
<dvt:area id="area" fillColor="#{tmapTargetActualBean.colorObj}"
          shortDesc="#{tmapTargetActualBean.tooltip}">
  <af:dragSource actions="COPY" discriminant="DnDDemoModel"/>
</dvt:area>
```

To use a map layer as a drop target:

1. In the Structure window, right-click the `areaLayer` component you are configuring as a drop target, and choose **Insert Inside Area Layer > Drop Target**.
2. Enter an expression for the `dropListener` that evaluates to a method on a managed bean that will handle the event.
3. In the managed bean referenced in the EL expression created in Step 2 for the `dropListener` attribute, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

The example below shows sample code for adding and configuring a map layer as a drop target.

```
<dvt:areaLayer id="areaLayer" layer="states">
  <af:dropTarget actions="COPY"

dropListener="#{TestDropHandler.handleCollectionFireDrop}">
  <af:dataFlavor flavorClass="java.util.Collection"/>
</af:dropTarget>
</dvt:areaLayer>
```

For more information about adding drag and drop functionality, see [Adding Drag and Drop Functionality for Components](#).

31

Using Hierarchy Viewer Components

This chapter describes how to use the ADF Data Visualization `hierarchyViewer` component to display data in hierarchy viewers using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the Fusion technology stack, then you can also use data controls to create hierarchy viewers. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About Hierarchy Viewer Components](#)
- [Using Hierarchy Viewer Components](#)
- [Managing Nodes in a Hierarchy Viewer](#)
- [Using Panel Cards](#)
- [Configuring Navigation in a Hierarchy Viewer](#)
- [Customizing the Appearance of a Hierarchy Viewer](#)
- [Adding Interactivity to a Hierarchy Viewer Component](#)
- [Adding Search to a Hierarchy Viewer](#)

About Hierarchy Viewer Components

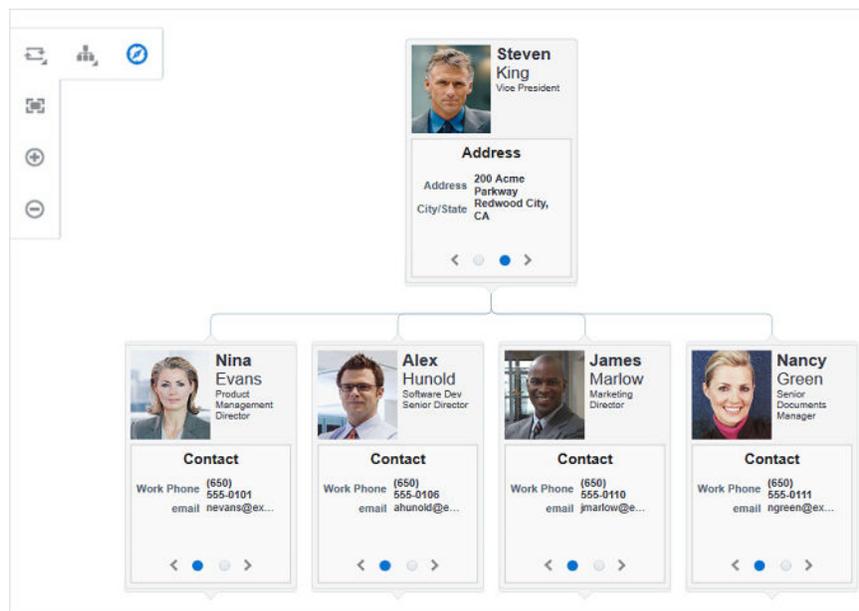
ADF DVT Hierarchy viewer components are used to display hierarchical data, i.e. data that contains master-detail relationships within itself. For example, you could create a hierarchy viewer that renders an organization chart from a data collection that contains information about the relationships between employees in an organization.

Hierarchy viewers use a shape called a node to reference the data in a hierarchy. The shape and content of the nodes is configurable, as well as the visual layout of the nodes. Nodes can display multiple views in a panel card.

Hierarchy Viewer Use Cases and Examples

A hierarchy viewer visually displays hierarchical data and the master-detail relationships. [Figure 31-1](#) shows a segment of a hierarchy viewer component at runtime that includes a control panel, a number of nodes, and links that connect the nodes. The nodes include a panel card that uses `af:showDetailItem` elements to display multiple sets of data.

Figure 31-1 Hierarchy Viewer Component with Control Panel and Nodes



End User and Presentation Features

The ADF Data Visualization hierarchy viewer component provides a range of features for end users, such as panning and zooming and changing the layout view. It also provides a range of presentation features, such as changing node shape, lines, and labels.

Layouts

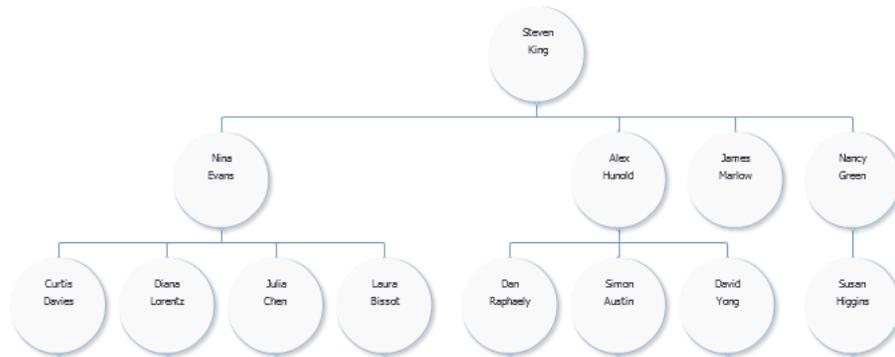
You can define the initial layout of the hierarchy viewer when you insert the component on the page from either the Data Controls panel to bind a data collection to the hierarchy viewer component or from the Components window to insert the component and bind to data later.

The layout of nodes in a hierarchy viewer is configurable and includes the following types of layouts:

- Vertical top down

[Figure 31-2](#) shows an example of a vertical top down layout.

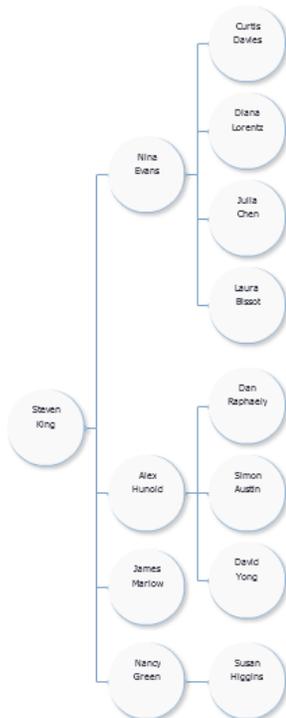
Figure 31-2 Hierarchy Viewer Vertical Top Down Layout



- Vertical bottom up
- Horizontal left-to-right

Figure 31-3 shows an example of a horizontal left-to-right layout.

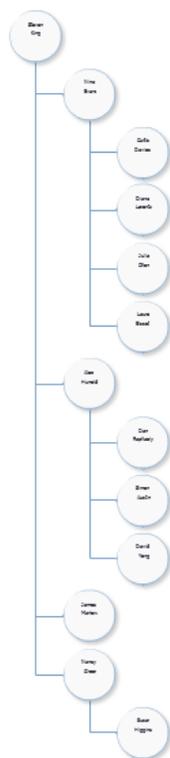
Figure 31-3 Hierarchy Viewer Horizontal Left-to-Right Layout



- Horizontal right-to-left
- Horizontal, direction depends on the locale
- Tree, indented tree

Figure 31-4 shows an example of a tree layout.

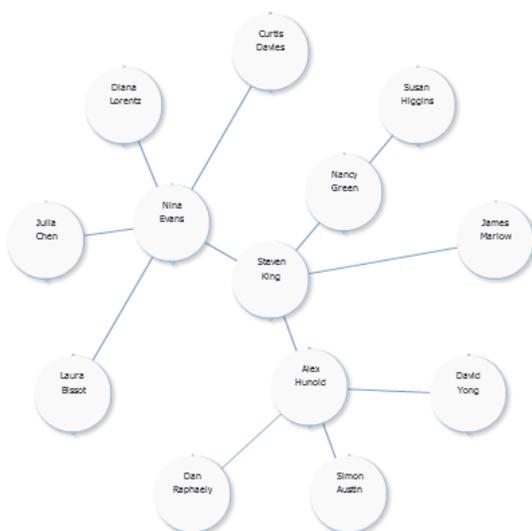
Figure 31-4 Hierarchy Viewer Tree Layout



- Radial, root node in center and successive child levels radiating outward from their parent nodes
- Circle, root node in center and all leaf nodes arranged in concentric circle, with parent nodes arranged within the circle

Figure 31-5 shows an example of a circle layout.

Figure 31-5 Hierarchy Viewer Circle Layout



Navigation

At runtime, the node contains controls that allow users to navigate between nodes and to show or hide other nodes by default. The end user uses the controls on the node to switch dynamically between the content that the panel cards reference.

At runtime, if a user double-clicks another node that has a value specified for its `setAnchorListener` property, that node becomes the anchor node.

At runtime, when a user moves the mouse over a node at zoom levels less than 76%, a hover window displaying node content at zoom level 100% is automatically displayed, allowing the user to see the full information regardless of zoom level. The controls on the hover window are active.

Panning

By default, panning in a hierarchy viewer is accomplished by clicking and dragging the component to reposition the view, or by using the panning control in the Control Panel.

You can disable the panning effect by setting the `panning` property to `none`.

Control Panel

The hierarchy viewer Control Panel provides tools for a user to manipulate the position and appearance of a hierarchy viewer component at runtime.

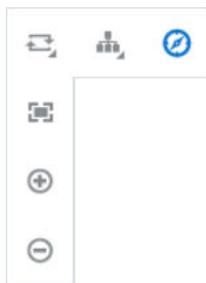
By default, it appears in a hidden state in the upper left-hand corner of the hierarchy viewer, as illustrated by [Figure 31-6](#).

Figure 31-6 Control Panel in Hidden State



Users click the **Hide or Show Control Panel** button shown in [Figure 31-6](#) to hide or expand the Control Panel. [Figure 31-7](#) shows the expanded Control Panel.

Figure 31-7 Control Panel in Show State



[Table 31-1](#) describes the functionality that the controls in the Control Panel provide to users. The Panel Selector is automatically enabled if a node in your hierarchy viewer

component contains a panel card with `af:showDetailItem` elements to display additional data. The Layout Selector appears automatically if the hierarchy viewer component uses one of the following layouts:

- Vertical top down
- Horizontal left to right
- Tree
- Radial
- Circle

Table 31-1 Elements in the Control Panel

Control	Name	Description
	Zoom to Fit	Allows user to zoom a hierarchy viewer component so that all nodes are visible within the viewport.
	Zoom Control	Allows user to zoom the hierarchy viewer component.
	Hide or Show	Hides or shows the Control Panel.
	Panel Selector	If you configured a panel card, displays the list of <code>af:showDetailItem</code> elements that you have defined. Users can use the panel selector to show the same panel on all nodes at once.

Table 31-1 (Cont.) Elements in the Control Panel

Control	Name	Description
	Layout Selector	Allows a choice of layouts. Users can change the layout of the hierarchy viewer component from the layout you defined to another one of the layout options presented by the component.

Printing

Hierarchy viewers are printed using the HTML view in the browser.

Bi-directional Support

Hierarchy viewers support bi-directional text in node content, the search panel, and the display of search results. Bi-directional text is text containing text in both text directionalities, both right-to-left (RTL) and left-to-right (LTR). It generally involves text containing different types of alphabets such as Arabic or Hebrew scripts.

Hierarchy viewers also provide bi-directional support for flipping panel cards from one node view to the next and for swapping the locations of the Control Panel and Search Panel if those elements are defined.

State Management

Hierarchy viewers support state management for user actions such as node selection, expansion, and lateral navigation. When a user selects a node, expands a node or navigates to the left or right within the same parent to view the next set of nodes, that state is maintained if the user returns to a page after navigating away, as in a tabbed panel.

State management is supported through hierarchy viewer attributes including `disclosedRowKeys`, `selectedRowKeys`, and `layout`.

Additional Functionality for Hierarchy Viewer Components

You may find it helpful to understand other ADF Faces features before you implement your hierarchy viewer component. Additionally, once you have added a hierarchy viewer component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that hierarchy viewer components can use:

- Partial page rendering: You may want a hierarchy viewer to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: Users can change the way the hierarchy viewer displays at runtime. Those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: You can make your hierarchy viewer components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Content Delivery: You can configure your hierarchy viewer to fetch a certain number of rows at a time from your data source using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- Skins and styles: You can customize the appearance of hierarchy viewers using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound hierarchy viewers based on how your ADF Business Components are configured. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using Hierarchy Viewer Components

To use the Hierarchy Viewer component, add the hierarchy viewer to a page using the Component Palette window. Then define the data for the hierarchy viewer and complete the additional configuration in JDeveloper using the tag attributes in the

Properties window. The hierarchy viewer component uses the same data model as the ADF Faces `tree` component.

A hierarchy viewer component requires data collections where a master-detail relationship exists between one or more detail collections and a master detail collection. You can test whether it is possible to bind a data collection to a hierarchy viewer component by first binding it to an ADF Faces `tree` component. If you can navigate the data collection using the ADF Faces `tree` component, it should be possible to bind it to a hierarchy viewer component.

When you add a hierarchy viewer component to a JSF page, JDeveloper adds a tree binding to the page definition file for the JSF page. For information about how to populate nodes in a tree binding with data, see *Using Trees to Display Master-Detail Objects* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

The data collections that you bind to nodes in a hierarchy viewer component must contain a recursive accessor if you want users to be able to navigate downward from the root node of the hierarchy viewer component. For information about navigating a hierarchy viewer component, see [Configuring Navigation in a Hierarchy Viewer](#).

Configuring Hierarchy Viewer Components

JDeveloper generates the following elements in JSF pages when you drag and drop components from the Components window onto a JSF page or when you use the Create Hierarchy Viewer dialog to create a hierarchy viewer component as described in the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

- Hierarchy viewer (`hierarchyViewer`): Wraps the node and link elements.
- Node (`node`): A node is a shape that references the data in a hierarchy, for example, employees in an organization or computers in a network. You configure the child elements of the `node` element to reference whatever data you want to display. The `node` element supports the use of one or more `f:facet` elements that display content at different zoom levels (100%, 75%, 50%, and 25%). The `f:facet` element supports the use of many ADF Faces components, such as `af:outputText`, `af:image`, and `af:panelGroupLayout`, in addition to the ADF Data Visualization `panelCard` component.

At runtime, the node contains controls that allow users to navigate between nodes and to show or hide other nodes by default. For information about specifying node content and defining zoom levels, see [How to Specify Node Content](#).

- Link (`link`): You set values for the attributes of the `link` element to connect one node with another node. For information about how to customize the appearance of the link and add labels, see [How to Configure the Display of Links and Labels](#).
- Panel card (`panelCard`): Provides a method to switch dynamically between multiple sets of content referenced by a node element using animation by, for example, horizontally sliding the content or flipping a node over.

The `f:facet` tag for each zoom level supports the use of a `dvt:panelCard` element that contains one or more `af:showDetailItem` elements defining the content to be displayed at the specified zoom level. At runtime, the end user uses the controls on the node to switch dynamically between the content that the

`af:showDetailItem` elements reference. For more information, see [Using Panel Cards](#).

 **Note:**

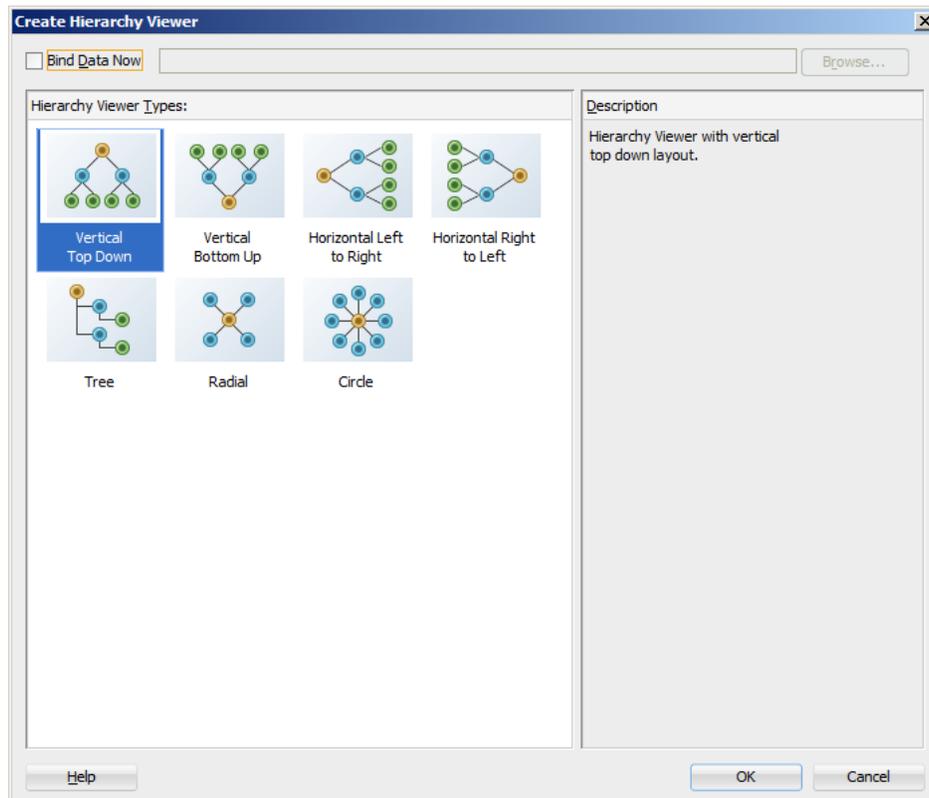
Unlike the other elements, the `dvt:panelCard` element is not generated if you choose the default quick layout option when using the Components window to create a hierarchy viewer.

How to Add a Hierarchy Viewer to a Page

You use the Components window to add a hierarchy viewer to a JSF page. When you drag and drop a hierarchy viewer component onto the page, the Create Hierarchy Viewer dialog displays available categories of hierarchy viewer layouts, with descriptions, to provide visual assistance when creating hierarchy viewers.

[Figure 31-8](#) shows the Create Hierarchy Viewer dialog for hierarchy viewers with the vertical top down layout type selected.

Figure 31-8 Create Hierarchy Viewer Dialog

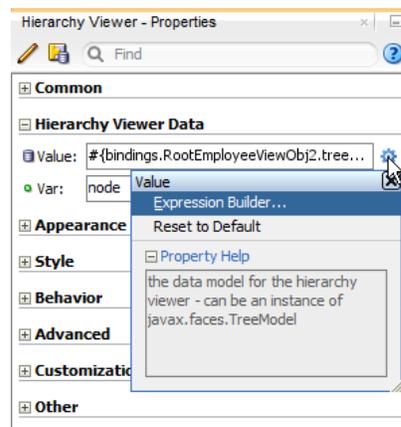


Once you select the hierarchy viewer layout, and the hierarchy viewer is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the hierarchy viewer. Alternatively, you can choose to

bind the data during creation and use the Properties window to configure additional display attributes.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. [Figure 31-9](#) shows the dropdown menu for a hierarchy viewer component `Value` attribute.

Figure 31-9 Hierarchy Viewer Value Attribute Dropdown Menu



Note:

If your application uses the Fusion technology stack, then you can use data controls to create a hierarchy viewer and the binding will be done for you. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a hierarchy viewer to a page:

1. In the ADF Data Visualizations page of the Components window, from the Hierarchy Viewer panel, drag and drop a **Hierarchy Viewer** onto the page to open the Create Hierarchy Viewer dialog.

Use the dialog to select the hierarchy viewer layout type. For help with the dialog, click **Help** or press F1.

2. In the Create Hierarchy Viewer dialog, click **OK** to add the hierarchy viewer to the page.

Optionally, use the dialog to bind the hierarchy viewer by selecting **Bind Data Now** and navigating to the ADF data control that represents the data you wish to display on the treemap. If you choose this option, the data binding fields in the dialog will be available for editing. For help with the dialog, press F1 or click **Help**.

3. In the Properties window, view the attributes for the hierarchy viewer. Use the **Help** button to display the complete tag documentation for the `hierarchyViewer` component.
4. Expand the **Common** section. Use this section to set the following attributes:
 - **Layout:** Specify the hierarchical layout of the hierarchy viewer. For a description with illustration of the valid values, see [Layouts](#).
 - **Ancestor Levels** (show sub-menu): Use to set the `displayLevelsAncestor` attribute that specifies the number of ancestor levels to display during initial render. This property is zero-based. A value of 0 means that no ancestor levels above the root will be shown. The default value is 0.
 - **Descendent Levels** (show sub-menu): Use to set the `displayLevelsChildren` attribute that specifies the number of child levels to display during initial render. This property is zero-based. A value of 0 means that no child levels below the root will be shown; the root itself will be shown. The default value is 1, which means that the root and the first level of children will be shown.

 **Note:**

You can also use the `disclosedRowKeys` attribute to specify the number of child levels to display during initial render. If you specify both `disclosedRowKeys` and `displayLevelsChildren` attributes, the `disclosedRowKeys` attribute takes precedence over `displayLevelsChildren`.

- **Nodes Per Level** (show sub-menu): Use to set the `levelFetchSize` attribute that specified the number of child nodes that will be fetched and displayed at a single time for each expanded parent node. Additional child nodes may be fetched and displayed by using the lateral navigation controls shown in the hierarchy viewer. The default value is 25.
5. Expand the **Hierarchy Viewer Data** section. Use this section to set the following attributes:
 - **Value:** Specify the data model for the hierarchy viewer; can be an instance of `javax.faces.TreeModel`.
 - **Var:** Specify the variable used to reference each element of the hierarchy viewer data collection. Once this component has completed rendering, this variable is removed or reverted back to its previous value.
 6. Expand the **Appearance** section. Use this section to set the following attributes:

- **Summary:** Enter a description of the hierarchy viewer. This description is accessed by screen reader users.
 - **EmptyText:** Specify the text to display when a hierarchy viewer does not display data.
7. Expand the **Behavior** section. Use this section to set the following attributes:
- **ControlPanelBehavior:** Specify the behavior of the Control Panel. For more information, see [How to Configure the Display of the Control Panel](#).
 - **Panning:** Specify panning behavior. The default value is `auto` for click and drag panning. You can also set it to `none` to disable panning.

What Happens When You Add a Hierarchy Viewer to a Page

When a hierarchy viewer component is inserted into a JSF page using the Create Hierarchy Viewer dialog, a set of child tags that support customization of the hierarchy viewer is automatically inserted. The hierarchy viewer component uses elements such as `af:panelGroupLayout`, `af:spacer`, and `af:separator` to define how content is displayed in the nodes.

The example below shows the code generated when the component is created by insertion from the Components window. Code related to the hierarchy viewer elements is highlighted in the example.

```
<dvt:hierarchyViewer id="hv1" layout="hier_vert_top" styleClass="AFStretchWidth">
  <dvt:link linkType="orthogonalRounded" id="l1"/>
  <dvt:node width="233" height="330" id="n1">
    <f:facet name="zoom100">
      <af:panelGroupLayout layout="vertical"
        styleClass="AFStretchWidth AFHVNodeStretchHeight AFHVNodePadding"
        id="pgl1">
        <af:panelGroupLayout layout="horizontal" id="pgl2">
          <af:panelGroupLayout styleClass="AFHVNodeImageSize" id="pgl3">
            <af:image source="#{null}" styleClass="AFHVNodeImageSize" id="i1"/>
          </af:panelGroupLayout>
          <af:panelGroupLayout>
            <af:spacer width="5" height="5" id="s1"/>
            <af:panelGroupLayout layout="vertical" id="pgl4">
              <af:outputText value=" attribute value1"
                styleClass="AFHVNodeTitleTextStyle" id="ot1"/>
              <af:outputText value=" attribute value2"
                styleClass="AFHVNodeSubtitleTextStyle" id="ot2"/>
              <af:outputText value=" attribute value3"
                styleClass="AFHVNodeTextStyle" id="ot3"/>
            </af:panelGroupLayout>
          </af:panelGroupLayout>
          <af:spacer height="5" id="s2"/>
          <af:separator id="s3"/>
          <af:spacer height="5" id="s4"/>
          <dvt:panelCard effect="slide_horz" styleClass="AFHVNodePadding" id="pc1">
            <af:showDetailItem text="first group title " id="sdi1">
              <af:panelFormLayout styleClass="AFStretchWidth AFHVNodeStretchHeight
                AFHVNodePadding"
                id="pfl1">
                <af:panelLabelAndMessage label="attribute label4"
                  styleClass="AFHVPanelCardLabelStyle" id="plam1">
                  <af:outputText value="attribute value4"
                    styleClass="AFHVPanelCardTextStyle" id="ot4"/>
                </af:panelLabelAndMessage>
                <af:panelLabelAndMessage label="attribute label5"
```

```

                styleClass="AFHVPanelCardLabelStyle" id="plam2">
        <af:outputText value="attribute value5"
                styleClass="AFHVPanelCardTextStyle" id="ot5"/>
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="attribute label6"
                styleClass="AFHVPanelCardLabelStyle" id="plam3">
        <af:outputText value="attribute value6"
                styleClass="AFHVPanelCardTextStyle" id="ot6"/>
    </af:panelLabelAndMessage>
</af:panelFormLayout>
</af:showDetailItem>
<af:showDetailItem text="second group title " id="sdi2">
    <af:panelFormLayout styleClass="AFStretchWidth AFHVNStretchHeight
AFHVNodePadding"
                id="pfl2">
        <af:panelLabelAndMessage label="attribute label7"
                styleClass="AFHVPanelCardLabelStyle" id="plam4">
            <af:outputText value="attribute value7"
                    styleClass="AFHVPanelCardTextStyle" id="ot7"/>
        </af:panelLabelAndMessage>
        <af:panelLabelAndMessage label="attribute label8"
                styleClass="AFHVPanelCardLabelStyle" id="plam5">
            <af:outputText value="attribute value8"
                    styleClass="AFHVPanelCardTextStyle" id="ot8"/>
        </af:panelLabelAndMessage>
        <af:panelLabelAndMessage label="attribute label9"
                styleClass="AFHVPanelCardLabelStyle" id="plam6">
            <af:outputText value="attribute value9"
                    styleClass="AFHVPanelCardTextStyle" id="ot9"/>
        </af:panelLabelAndMessage>
    </af:panelFormLayout>
</af:showDetailItem>
</dvt:panelCard>
</af:panelGroupLayout>
</f:facet>
</dvt:node>
</dvt:hierarchyViewer>

```

What You May Need to Know About Hierarchy Viewer Rendering and Image Formats

By default, the hierarchy viewer component renders in HTML5. When HTML5 is not supported by the browser and Flash 10 or higher is available on the client, the hierarchy viewer is rendered in a Flash Player. If HTML5 and Flash10 or higher are not available, the hierarchy viewer is rendered in HTML4. While HTML4 rendering follows HTML5 and Flash rendering as closely as possible, there are some differences.

For the most part, hierarchy viewer display and features are supported with the following exceptions:

- Isolate and restore nodes is not available.
- Node shapes are limited to rectangular.
- For links, the link end connector is not supported, link type is limited to orthogonal, and link style is limited to a solid line.
- For the control panel, all panel cards cannot be switched, panning is limited to scroll bars, and zooming and zoom to fit is limited to four zoom facets.

- Search is not supported.
- Emailable page is not supported.
- Node detail hover window is not supported.

Managing Nodes in a Hierarchy Viewer

A node is a shape that represents the individual elements in an ADF DVT hierarchy viewer component at runtime. Examples of individual elements in a hierarchy viewer component include an employee in an organization chart or a computer in a network diagram.

By default, each node in a hierarchy viewer component includes controls that allow users to do the following:

- Navigate to other nodes in a hierarchy viewer component.

The top of each node contains a single **Isolate** or **Restore** button. The **Isolate** button allows the user to reduce the hierarchy temporarily to the chosen node and its displayed children. Users click **Restore** to return the hierarchy to the original view.

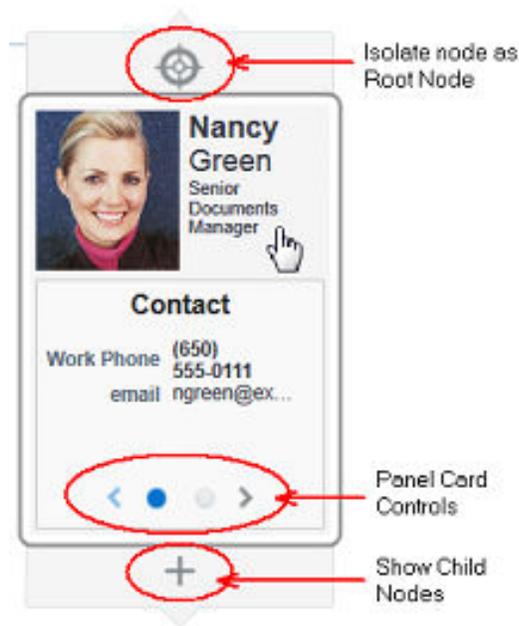
- Show or hide child nodes of the currently selected node in a hierarchy viewer component.

The single **Show or Hide** button appears on the bottom of every node. When a user clicks one of these icons, the page displays or hides the node's children if they exist, and the component generates a `RowDisclosureEvent` event. You can register a custom `rowDisclosureListener` method to handle any processing in response to the event in the same way as an `af:tree` component. For more information, see [What You May Need to Know About Programmatically Expanding and Collapsing Nodes](#).

If you use a panel card to display different sets of information for the node that the hierarchy viewer component references, controls at the bottom of the node allow the user to change the information set in the active node. For more information, see [Using Panel Cards](#).

[Figure 31-10](#) shows an example of a node with controls that allow an end user to isolate the node as the anchor node, show the child nodes, and change the node to show different sets of information in the active node. For information about how to configure the controls on a node, see [How to Configure the Controls on a Node](#).

Figure 31-10 Hierarchy Viewer Node Controls



How to Specify Node Content

Although a node contains controls by default that allow you to navigate to a node and show or hide nodes, nodes do not by default include content unless you used a quick start layout when creating the hierarchy viewer component. You must define what content a node renders at runtime.

You can specify node content when you associate data bindings with the hierarchy viewer component as described in the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*. You can also specify content that is stored in a managed bean.

By default, a hierarchy viewer component that you create contains one node with one facet element that has a zoom level of 100%:

```
<f:facet name="zoom100" />
```

You can insert three more instances of the facet element into the hierarchy viewer component with the following zoom levels:

- 25%: zoom25
- 50%: zoom50
- 75%: zoom75

Use these zoom level definitions to improve readability of node content when the hierarchy viewer is zoomed out to display more nodes and less display room is available in each node. You can define a subset of the available data collection within one or more of the facet elements. For example, if you have a data collection with node attributes that reference data about a company department such as its name,

location, and number, you can specify a facet element with a zoom level of 50% that references the node attribute for just the department's name and number.

At runtime, when a user moves the mouse over a node at any zoom level less than 76%, a hover window displaying node content at zoom level 100% is automatically displayed, allowing the user to see the full information regardless of zoom level. The controls on the hover window are active.

Each of the facet elements that you insert can be used to reference other components. You can use one or more of the following ADF Faces components when you define content for a node in a hierarchy viewer component. The node component's facet support the following components:

- Layout components including: `af:panelFormLayout`, `af:panelGroupLayout`, `af:separator`, `af:showDetailItem`, and `af:spacer`. For more information about using these components, see [Organizing Content on Web Pages](#).
- Menu components including: `af:menu` and `af:menuItem`. For more information about these components, see [Using Menus, Toolbars, and Toolboxes](#).
- Output components including: `af:outputFormatted` and `af:outputText`. For more information about these components, see [Using Output Components](#).
- Message component `af:panelLabelAndMessage`. For more information about this component, see [Displaying Tips, Messages, and Help](#).
- Navigation components including: `af:button`, `af:link`, and `af:commandMenuItem`. For more information about these components, see [Working with Navigation Components](#).
- Image component `af:image`. For information about how to use the `af:image` component, see [Including Images in a Hierarchy Viewer](#).
- `af:showPopupBehavior`: For information about how to use the `af:showPopupBehavior` component, see [Configuring a Hierarchy Viewer to Invoke a Popup Window](#).
- Hierarchy viewer child component `dvt:panelCard`: For information about how to use the `dvt:panelCard` component, see [Using Panel Cards](#).

 **Note:**

Unsupported components are flagged at design time.

By default, the hierarchy viewer component renders in HTML5. When HTML5 is not available, the hierarchy viewer will render in Flash or HTML4, and certain properties that you specify as node content may not be supported. For more information, see [What You May Need to Know About Hierarchy Viewer Rendering and Image Formats](#).

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To add a node to a hierarchy viewer component:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Insert Inside Hierarchy Viewer > Node**.

The following entry appears in the JSF page:

```
<dvt:node>
  <f:facet name="zoom100"/>
</dvt:node>
```

2. In the Structure window, right-click **dvt:node** and choose **Go to Properties**.
3. Configure the appropriate properties in the Properties window.

For example, set a value for the **Type** property to associate a node component with an accessor. The following code appears in the JSF page if you associate `model.HvtestView` with the node:

```
<dvt:node type="model.HvtestView"/>
```

For more information, see [Specifying a Node Definition for an Accessor](#).

How to Configure the Controls on a Node

The node component (`node`) exposes a number of properties that allow you to determine if controls such as Restore, Isolate, Show, or Hide appear at runtime. It also exposes properties that determine the size and shape of the node at runtime.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer node on your page. If you do not, follow the instructions in this chapter to add a node to a hierarchy viewer. For more information, see [How to Specify Node Content](#).

To configure the controls on a node:

1. In the Structure window, right-click **dvt:node** and choose **Go to Properties**.
2. In the Properties window, in the **Appearance** section, configure properties for the node, as described in [Table 31-2](#).

Table 31-2 Node Configuration Properties

To do this:	Set the following value for this property:
Configure the Hide or Show controls to appear or not on a node.	Set <code>showExpandChildren</code> to <code>False</code> to hide the controls. By default the property is set to <code>True</code> .

Table 31-2 (Cont.) Node Configuration Properties

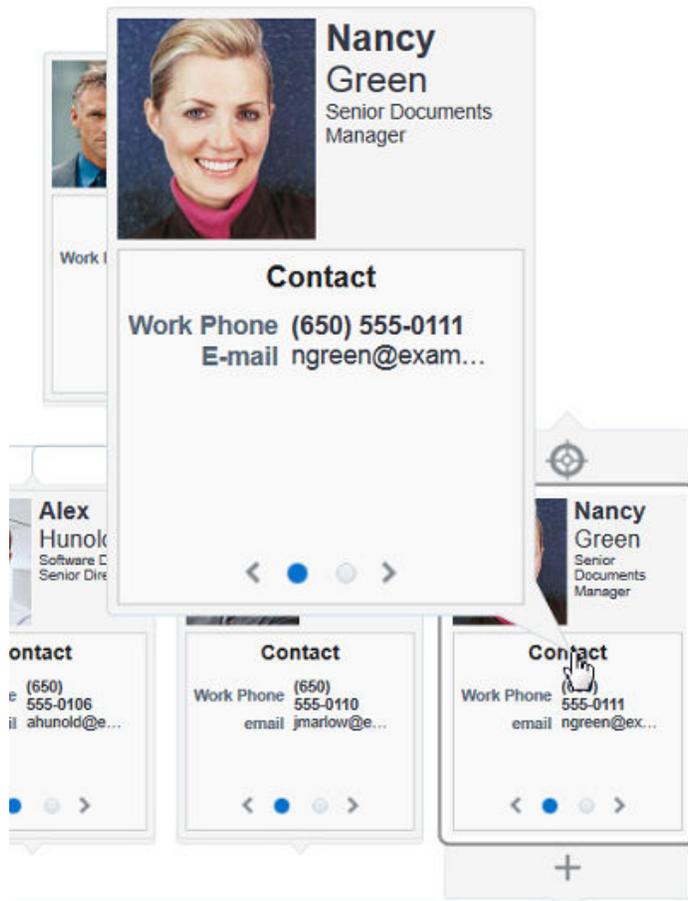
To do this:	Set the following value for this property:
Configure the Restore or Isolate controls to appear or not on the node.	Set the <code>showIsolate</code> property to <code>False</code> to hide these controls on the node. By default the property is set to <code>true</code> .
Configure the Navigate Up control to appear or not on the node.	Set the <code>showNavigateUp</code> property to <code>False</code> to hide this control on the node. By default the property is set to <code>true</code> . If the <code>showNavigateUp</code> property is set to <code>true</code> , for the control to render, you must also set a value for the hierarchy viewer component's <code>navigateUpListener</code> property, as described in How to Configure Upward Navigation in a Hierarchy Viewer .
Configure the height and width of a node.	Set values for the <code>width</code> and <code>height</code> properties.
Select the shape of the node.	Select a value from the Shape dropdown list. Available values are: <ul style="list-style-type: none"> • <code>ellipse</code> • <code>rect</code> • <code>roundedRect</code> (default)

- For information about configuring the properties in the Style section of the Properties window for the node component, see [Changing the Style Properties of a Component](#).

The hover detail window is automatically displayed when the user moves the mouse over the node at zoom levels less than 76%, reflecting the `shape` attribute set for the node. A node with the `shape` attribute `roundedRect`, for example, will have a hover window with the same attribute, as shown in [Figure 31-11](#).

You can disable the display of the detail window when hovering a node that is not at the 76-100% zoom level. For more information, see [How to Disable the Hover Detail Window](#).

Figure 31-11 Hover Window in Hierarchy Viewer Node



Specifying a Node Definition for an Accessor

By default, you associate a node component with an accessor when you use the Create Hierarchy Viewer dialog to create a hierarchy viewer component, as described in the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*. The Create Hierarchy Viewer dialog sets the node component's `type` property to a specific accessor.

You can configure a node component's `type` property to use one or more specified accessors. Alternatively, you can configure a node component's `rendered` property to use a node definition across accessors, as described in [Associating a Node Definition with a Particular Set of Data Rows](#). When the hierarchy viewer component determines which node definition to use for a particular data row, it first checks for a match on the `type` property:

- If the `type` property matches and the `rendered` property value is `true` (default), the hierarchy viewer component uses the node definition.
- If the `type` property does not match, the hierarchy viewer component uses the first node definition whose `rendered` property evaluates to `true`. The result of evaluating the `rendered` property does not affect the `type` property.

Associating a Node Definition with a Particular Set of Data Rows

You can use a node component's `rendered` property to associate the node with a particular set of data rows or with a single data row. The `rendered` property accepts a boolean value so you can write an EL expression that evaluates to `true` or `false` to determine what data rows you associate with a node definition.

For example, assume that you want a node to display data based on job title. You write an EL expression for the node component's `rendered` property similar to the following pseudo EL expression that evaluates to `true` when a job title matches the value you specify (in this example, CEO):

```
rendered="#{node.title == 'CEO'}"
```

When you use the node component's `rendered` property in this way, you do not define a value for the node component's `type` property.

 **Note:**

The hierarchy viewer will use the first node definition whose `rendered` property evaluates to `true`. The order of the hierarchy viewer's node definitions is important.

How to Specify Ancestor Levels for an Anchor Node

The anchor node of a hierarchy viewer component is the root of the hierarchy returned by the tree binding. Depending on the use case, there can be multiple root nodes, for example, a hierarchy viewer component that renders an organization chart with one or more managers. When a hierarchy viewer component renders at runtime, the node that has focus is the anchor node. If a user double-clicks another node at runtime that has a value specified for its `setAnchorListener` property, that node becomes the anchor node.

You can configure the hierarchy viewer to display one or more levels above the anchor node, the ancestor levels. For example, if you search for an employee in a company, you may wish to display the chain of management above the employee. Specify ancestor levels using the `displayLevelsAncestor` property.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to add a hierarchy viewer to your page. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To specify the number of ancestor levels for an anchor node:

1. In the Structure window, right-click the `dvt:hierarchyViewer` node and choose **Go to Properties**.
2. Expand the **Common** section of the Properties window.
3. In the **Ancestor Levels** field, specify the number of levels of ancestor nodes that you want to appear at runtime.

For example, the following entry appears in the JSF page if you entered 2 as the number of ancestor levels for the anchor node.

```
displayLevelsAncestor="2"
```

4. Save changes to the JSF page.

Using Panel Cards

You can use the ADF panel card component in conjunction with the ADF DVT hierarchy viewer component to hold different sets of information for the nodes that the hierarchy viewer component references. The panel card component is an area inside the node element that can include one or more `af:showDetailItem` elements.

Each of the `af:showDetailItem` elements references a set of content. For example, a hierarchy viewer component that renders an organization chart would include a node for employees in the organization. This node could include a panel card component that references contact information using an `af:showDetailItem` element and another `af:showDetailItem` element that references salary information.

A panel card component displays the content referenced by one `af:showDetailItem` element at runtime. The panel card component renders navigation buttons and other controls that allow the user to switch between the sets of data referenced by `af:showDetailItem` elements. The controls that allow users to switch between different sets of data can be configured with optional transitional effects. For example, you can configure a panel card to horizontally slide between one set of data referenced by an `af:showDetailItem` element to another set of data referenced by another `af:showDetailItem` element.

How to Create a Panel Card

You can insert a panel card component into the JSF page that renders the hierarchy viewer component by using the context menu that appears when you select the Facet zoom element in the Structure window for the JSF page.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To create a panel card:

1. In the Structure window, right-click the zoom level within the node where you want to create a panel card.

For example, select **f:facet - zoom100**.

2. If the selected facet does not already contain a `panelGroupLayout` component, select **Insert Inside zoomLevel > Panel Group Layout** to create a container for the panel card.
3. Use the Properties window to configure the properties of the panel group layout. For more information about configuring panel group layout components, see [How to Use the panelGroupLayout Component](#).
4. In the Structure window, right-click the **af:panelGroupLayout** node and select **Insert Inside Panel Group Layout > Panel Card**.
5. Use the Properties window to configure the panel card's properties.

For example, set a value for the **Effect** property in the Advanced properties for the panel card component to specify the effect used when transitioning between content on the panel card. Valid values are:

- `slideHorz` (default)
Old content slides out on one side while new content slides in from the other side.
- `immediate`
Content displays immediately with no transition effect.
- `flipHorz`
The **showDetailItem** flips over to reveal new contents.
- `nodeFlipHorz`
The entire node flips over to reveal new contents.
- `cubeRotateHorz`
The **showDetailItem** rotates as if on the face of a cube to reveal new contents.
- `nodeCubeRotateHorz`
The entire node rotates as if on the face of a cube to reveal new contents.

 **Note:**

Valid values also include `slide_horz`, `flip_horz`, `node_flip_horz`, `cube_rotate_horz`, and `node_cube_rotate_horz`. These values are deprecated in favor of the mixed case values, but you may still see them in use in older code and documentation.

6. In the Structure window, right-click **dvt:panelCard** and choose **Insert Inside Panel Card > Show Detail Item**.
7. Use the Properties window to configure the properties of the `af:showDetailItem` element. For help with the Properties window, click **Help** or press F1.

8. Add elements to the `af:showDetailItem` element to display the desired content. In the Structure window, right-click **af:showDetailItem** and choose the element to insert.

For example, if your hierarchy viewer renders an organization chart, you could add an element to display the employee's ID. In the Structure window, right-click **af:showDetailItem** and choose **Insert Inside Show Detail Item > ADF Faces > Output Text**.

9. Use the Properties window to configure the properties for the elements you added to the `af:showDetailItem` element in Step 8. For help with the Properties window, click **Help** or press F1.
10. Repeat Step 6 through Step 9 for each set of content that you want the panel card to display.

What Happens at Runtime: How the Panel Card Component Is Rendered

At runtime, a node appears and displays one panel card component. Users can click the navigation buttons at the bottom of the panel card to navigate to the next set of content referenced by one of the panel card's `af:showDetailItem` child elements.

Figure 31-12 shows a node with a panel card component where two different `af:showDetailItem` child elements reference different sets of information (Contact and Address). The controls in the example include arrows to slide through the panel cards as well as buttons to directly select the panel card to display. Tooltips display for both control options.

Figure 31-12 Runtime View of a Node with a Panel Card



Configuring Navigation in a Hierarchy Viewer

By default, an ADF DVT hierarchy viewer component has downward navigation configured for root and inner nodes. You can configure the hierarchy viewer

component to enable upward navigation and to determine the number of nodes to appear when a user navigates between nodes on the same level.

For more information about node types, see [Managing Nodes in a Hierarchy Viewer](#).

How to Configure Upward Navigation in a Hierarchy Viewer

If you want to configure upward navigation for a hierarchy view component, you configure a value for the hierarchy viewer component's `navigateUpListener` property.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To configure upward navigation for a hierarchy viewer component:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section of the Properties window and write a value in the **Navigate Up** field for the hierarchy viewer component's `navigateUpListener` property that specifies a method to update the data model so that it references the new anchor node when the user navigates up to a new anchor node.

If you need help specifying a value, choose Method Expression Builder from the **Navigate Up** dropdown menu to enter the Method Expression Builder dialog. For help with the Method Expression Builder dialog, click **Help** or press F1.

3. Save the page.

How to Configure Same-Level Navigation in a Hierarchy Viewer

Same-level navigation between the nodes in a hierarchy viewer component is enabled by default. You can configure the hierarchy viewer component to determine how many nodes to display at a time.

When you do this, controls appear that enable users to navigate to the following:

- Left or right to view the next set of nodes
- First or last set of nodes in the collection of available nodes

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To configure same-level navigation in a hierarchy viewer component:

1. In the Structure window, right-click the `dvt:hierarchyViewer` node and choose **Go to Properties**.
2. Expand the **Common** section of the Properties window and specify the number of nodes that you want to appear at runtime in the **Nodes Per Level** field (`levelFetchSize`).

For example, the following entry appears in the JSF page if you entered 3 as the number of nodes:

```
levelFetchSize="3"
```

3. Save the page.

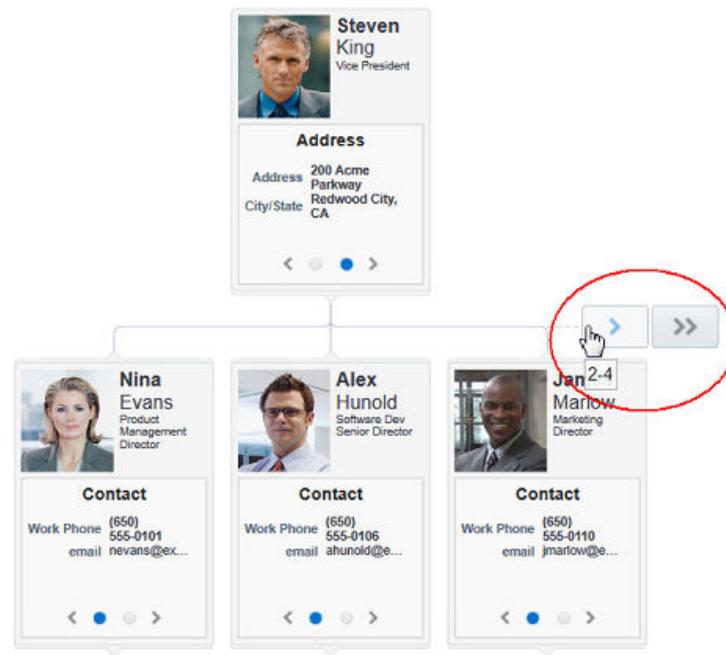
What Happens When You Configure Same-Level Navigation in a Hierarchy Viewer

At runtime, the hierarchy viewer component renders the number of nodes that you specified as a value for the hierarchy viewer component's `levelFetchSize` property. It also renders controls that allow users to do the following:

- Navigate to the left or right to view the next set of nodes
- Navigate to the first or last set of nodes in the collection of available nodes

[Figure 31-13](#) shows a runtime example where `levelFetchSize="3"`. When a user moves the mouse over the control, as shown in the circled area in [Figure 31-13](#), the control that allows users to navigate to the last set of nodes appears.

Figure 31-13 Hierarchy Viewer Component with Same-Level Navigation



Customizing the Appearance of a Hierarchy Viewer

You can customize the ADF DVT hierarchy viewer component size and appearance including adding images, configuring the display of the control panel and hover detail window, and customizing links and labels.

You can change the appearance of your hierarchy viewer component by changing skins and component style attributes, as described in [Customizing the Appearance Using Styles and Skins](#).

How to Adjust the Display Size and Styles of a Hierarchy Viewer

You can configure the hierarchy viewer's size and style using the `inlineStyle` and `styleClass` attributes. Both attributes are available in the **Style** section in the Properties window for the `dvt:hierarchyViewer` or `dvt:node` component. Using these attributes, you can customize stylistic features such as fonts, borders, and background elements.

The `styleClass` attribute is a CSS style class selector used to group a set of inline styles. The style classes can be defined using an EL Expression that evaluates to a style class at runtime. You can also specify an ADF public style class. For example, you can use `AFHVNodeImageSize` to set the size of an image to fit inside a hierarchy viewer.

The `inlineStyle` attribute is a list of CSS styles, separated by semicolons, that can set individual style attributes. For example, you can specify `color:blue;font-style:italic` to change the color and font style of an `af:outputText` component.

For additional information about using the `styleClass` and `inlineStyle` attributes, see [Changing the Style Properties of a Component](#).

The page containing the hierarchy viewer may also impose limitations on the ability to change the size or style. For more information about page layouts, see [Organizing Content on Web Pages](#).

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To adjust the size or style of a hierarchy viewer:

1. In the Structure window for the JSF page that contains the hierarchy viewer component, right-click the **dvt:hierarchyViewer** node and choose **Go to Properties**.
2. To adjust the size or style of the hierarchy viewer using inline styles, in the Properties window, expand the **Style** section and specify the following values for the **InlineStyle** property:
 - `width`
Write a value in percent (%) or pixels (px). The default value for width is 100%.
 - `height`
Write a value in percent (%) or pixels (px). The default value for height is 600px.

The final value that you enter for **InlineStyle** must use this syntax:

```
width:100%;height:600px;
```

3. To adjust the size or style of the hierarchy viewer using style classes, in the Properties window, specify a value for the **StyleClass** property.
For example, to specify 100% for the height of the hierarchy viewer, enter the following for **StyleClass**: `AFStretchHeight`.
4. Save changes to the JSF page.

Including Images in a Hierarchy Viewer

You can configure a hierarchy viewer component to display images in the nodes of a hierarchy viewer component at runtime. This can be useful where, for example, you want pictures to appear in an organization chart.

Insert an `af:image` component with the source attribute bound to the URL of the desired image. The following code example renders an image.

```
<af:panelGroupLayout>  
  <af:image source="/person_id=#{node.PersonId}"
```

```
        shortDesc="Employee Image"  
        styleClass="AFHVNodeImageSize" />  
</af:panelGroupLayout>
```

For more information about the `af:panelGroupLayout` component, see [How to Use the `panelGroupLayout` Component](#).

How to Configure the Display of the Control Panel

Although you cannot configure the Control Panel to appear in another location, you can configure the hierarchy viewer component to act as follows when the hierarchy viewer component renders at runtime:

- Appears in an expanded or show state
- Appears in a collapsed or hidden state
- Does not appear to users

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To configure the display of the Control Panel:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and choose one of the following values from the **ControlPanelBehavior** dropdown list:
 - `hidden`
Select this value if you do not want the Control Panel to appear at runtime.
 - `initCollapsed`
This is the default value. The Control Panel appears in a collapsed or hidden state at runtime.
 - `initExpanded`
Select this value if you want the Control Panel to appear in an expanded or show state at runtime.
3. Save changes to the JSF page.

How to Configure the Display of Links and Labels

In a hierarchy viewer the relationships between nodes are represented by lines that link the nodes. The links can be configured to include labels.

Figure 31-14 illustrates links and labels in a hierarchy viewer.

Figure 31-14 Hierarchy Viewer Links and Labels



You can customize the appearance of links and labels by setting properties of the `dvt:link` element in a hierarchy viewer. Figure 31-15 illustrates links with a `dashDot` value set for the `linkStyle` attribute.

Figure 31-15 Links with dashDot Link Style



Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create one. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To customize the display of links and labels:

1. In the Structure window, right-click the **dvt:link** node and choose **Go to Properties**.
2. In the Properties window, set the following attributes to customize the appearance of links between nodes in a hierarchy viewer as desired:
 - **LinkStyle**: Sets the style of the link, for example, dotted or dashed line.
 - **LinkColor**: Sets the color of the link.
 - **LinkWidth**: Sets the width of the link, in pixels.
 - **LinkType**: Sets the type of link, for example, direct line or smooth curved line fitted to what would have been a single right angle.
 - **EndConnectorType**: Sets the style of the link connection end to `none` (default) or `arrowOpen`.

3. Also in the Properties window, enter text for the label associated with the link in the **Label** property.

Alternatively, specify an EL expression to stamp out the link label based on the child node. For example, write an EL expression similar to the following where the node `var` attribute refers to the child node associated with the link.

```
label="{node.relationship}"
```

For more information about creating EL expressions, see [Creating EL Expressions](#).

4. Optionally, also in the Properties window, use the **Rendered** property to stamp the link for a particular relationship between nodes. The property accepts a boolean value so you can write an EL expression that evaluates to `true` or `false` to determine if the link represents the relationship. For example, assume that you want a link to display based on reporting relationship. You write an EL expression for the link component's **Rendered** property similar to the following EL expression that evaluates to `true` when the relationship matches the value you specify (in this example, `CONSULTANT`):

```
rendered="#{node.relationship == "CEO"}
```

How to Disable the Hover Detail Window

By default, the hover window automatically displays when the zoom level is below 76%. If your hierarchy viewer uses popups, the hover window can interfere with the popup display. You can use the hierarchy viewer `detailWindow` attribute to turn off the display of the hover window.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For information, see [How to Add a Hierarchy Viewer to a Page](#).

To disable the hierarchy viewer hover window:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section and select one of the following values from the **DetailWindow** dropdown list:
 - `auto`
This is the default value. The hover window is always enabled.
 - `none`
Select this value if you do not want to enable the hover window.

What You May Need to Know About Skinning and Customizing the Appearance of a Hierarchy Viewer

Hierarchy viewers also support skinning to customize the color and font styles for the top level components as well as the nodes, buttons, and links. In addition, you can use skinning to define the styles for a hierarchy viewer when the user hovers the mouse over or selects a navigation button.

The example below shows the skinning key for a hierarchy viewer configured to show the border color of the panel card's navigation button in black when the user selects it.

```
af|dvt-panelCard::navigation-button:active
{
  -tr-border-color:#000000;
}
```

For the complete list of hierarchy viewer skinning keys, see the *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces Skin Selectors*. For additional information about customizing your application using skins, see [Customizing the Appearance Using Styles and Skins](#).

Adding Interactivity to a Hierarchy Viewer Component

You can configure an ADF DVT hierarchy viewer component to invoke popup windows, display menus with functionality and data from other pages in your Oracle Fusion web application, or support drag and drop functionality.

How to Configure Node Selection Action

By default, clicking a hierarchy viewer node at runtime selects the node. You can customize this interaction by setting the `clickBehavior` attribute on the `dvt:node` component.

Valid values for this property include:

- `focus`: The node receives focus and is selected when clicked (default).
- `expandCollapse`: Child node elements are either expanded or collapsed, depending on their current expansion state.
- `isolateRestore`: The node is either isolated or restored, depending on its current state.
- `none`: Clicking the node does nothing.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For information, see [How to Add a Hierarchy Viewer to a Page](#).

To configure node selection action:

1. In the structure window, right-click the **dvt:node** node and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section and choose the value for **ClickBehavior** from the attribute's dropdown list.
3. Save changes to the JSF page.

Configuring a Hierarchy Viewer to Invoke a Popup Window

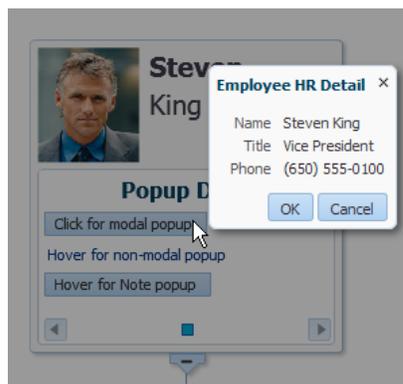
You can invoke a popup window from a hierarchy viewer node by specifying values for the `af:showPopupBehavior` tag and invoking it from a command component, for example, `af:button`. You must nest the command component that invokes the popup inside an `f:facet` element in a node of the hierarchy viewer component.

The `triggerType` property of an `af:showPopupBehavior` tag used in this scenario supports only the following values:

- `action`
- `mouseHover`

For example, [Figure 31-16](#) shows a modal popup invoked from an **HR Detail** link in the node. The example below shows sample code for creating the popup.

Figure 31-16 Modal Popup in Hierarchy Viewer Node



```
<af:popup id="popupDialog" contentDelivery="lazyUncached"
eventContext="launcher"
    launcherVar="source">
    <af:setPropertyListener from="#{source.currentRowData}"
        to="#{myBean.selectedEmployee}"
type="popupFetch"/>
    <af:dialog title="Employee HR Detail">
        <af:panelFormLayout>
            <af:panelLabelAndMessage label="Name" >
                <af:outputText value="#{myBean.selectedEmployee.firstName}
```

```

#{myBean.selectedEmployee.lastName}"/>
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="Official Title" >
        <af:outputText value="#{myBean.selectedEmployee.officialTitle}"/>
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="HR Manager Id" >
        <af:outputText value="#{myBean.selectedEmployee.hrMgrPersonId}"/>
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="HR Rep Id" >
        <af:outputText value="#{myBean.selectedEmployee.hrRepPersonId}"/>
    </af:panelLabelAndMessage>
</af:panelFormLayout>
</af:dialog>
</af:popup>

```

The example below shows sample code for the invoking the popup from a hierarchy viewer component. For brevity, elements such as `<af:panelGroupLayout>`, `<af:spacer>`, and `<af:separator>` are not included in the sample code.

```

<f:facet name="zoom100">
    ...
    <dvt:panelCard effect="slideHorz"
        ...
        <af:showDetailItem text="Contact "
            ...
            <af:button text="Show HR Detail"
                inlineStyle="font-size:14px;color:#383A47"
                id = bul>
                <af:showPopupBehavior popupId=":popupDialog" triggerType="action"
                    align="endAfter" alignId="bul" />
            </af:button>
        </showDetailItem>
    </dvt:panelCard>
</f:facet>

```

For more information about using the `af:showPopupBehavior` tag, see [Declaratively Invoking a Popup](#).

Configuring Hierarchy Viewer Drag and Drop

Hierarchy Viewers support a variety of drag and drop scenarios between components.

Hierarchy viewers support the following drag and drop scenarios:

- Drag and drop one or more nodes within a hierarchy viewer
- Drag one or more nodes from a hierarchy viewer to another component
- Drag one or more items from another component to a hierarchy viewer

[Figure 31-17](#) shows a hierarchy viewer configured to allow drags and drops within itself. In this example, if you click a node, you can drag it to the background to make it another root in the hierarchy or drag it to another node to add it as a child of that node.

Figure 31-17 Hierarchy Viewer Showing a Node Drag

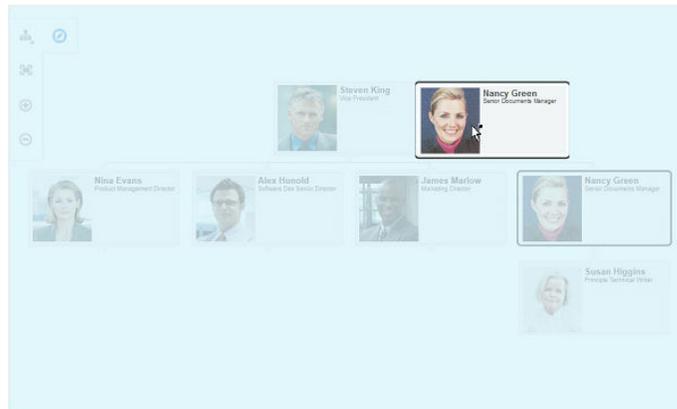
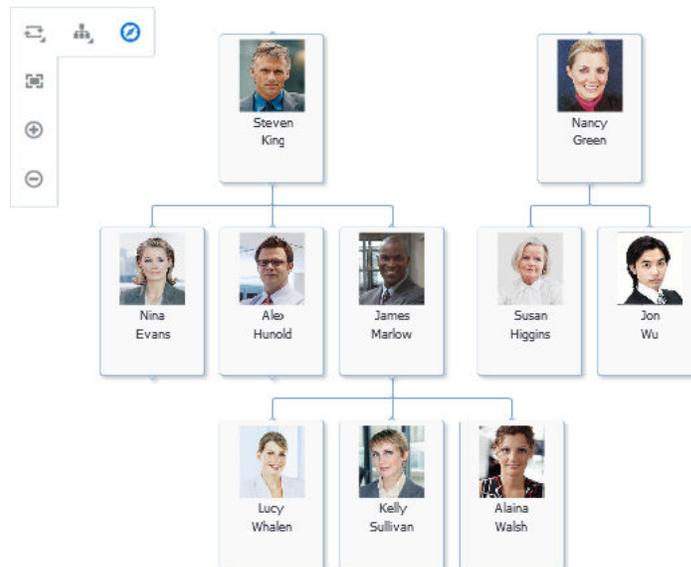


Figure 31-18 shows the result of a drag to the hierarchy viewer background. Nancy Green and her subordinates are now shown as a new tree in the hierarchy.

Figure 31-18 Hierarchy Viewer After Node Drag to Background



If you drag the node to another node, the dragged node and its children become the child of the targeted node. Figure 31-19 shows the result of the drag to the node containing the data for Nina Evans. Nancy Green and her subordinates are now shown as subordinates to Nina Evans.

Figure 31-19 Hierarchy Viewer After Node Drag to Another Node

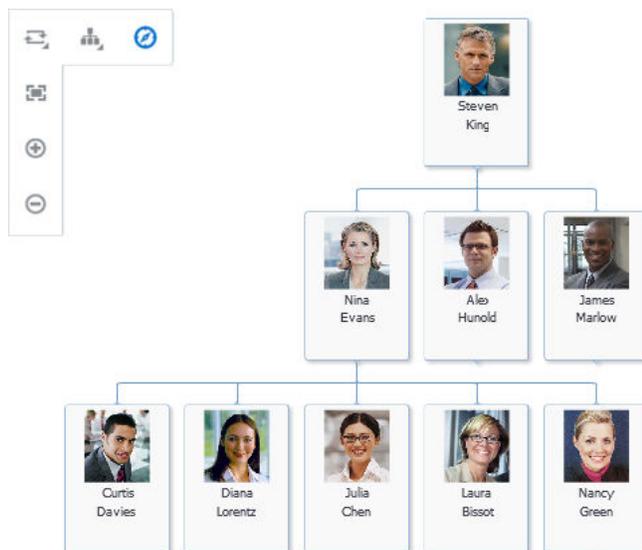


Figure 31-20 shows an example of the same hierarchy viewer configured to allow drops to or drags from an `af:outputFormatted` component. In this example, the user can drag one or more nodes to the drop text, and the text will change to indicate which node(s) the user dragged and which operation was performed. If the user drags from the drag text to a hierarchy viewer node or background, the text will change to indicate where the text was dragged and which operation was performed.

Figure 31-20 Hierarchy Viewer Configured for Drag and Drop to Another Component

Demonstrates Drag and Drop Features

- Drag a node onto the 'Drop a node here' text
- Drag the 'Drag this text onto a node' text onto a node

Drop a node here

Drag this text onto a node

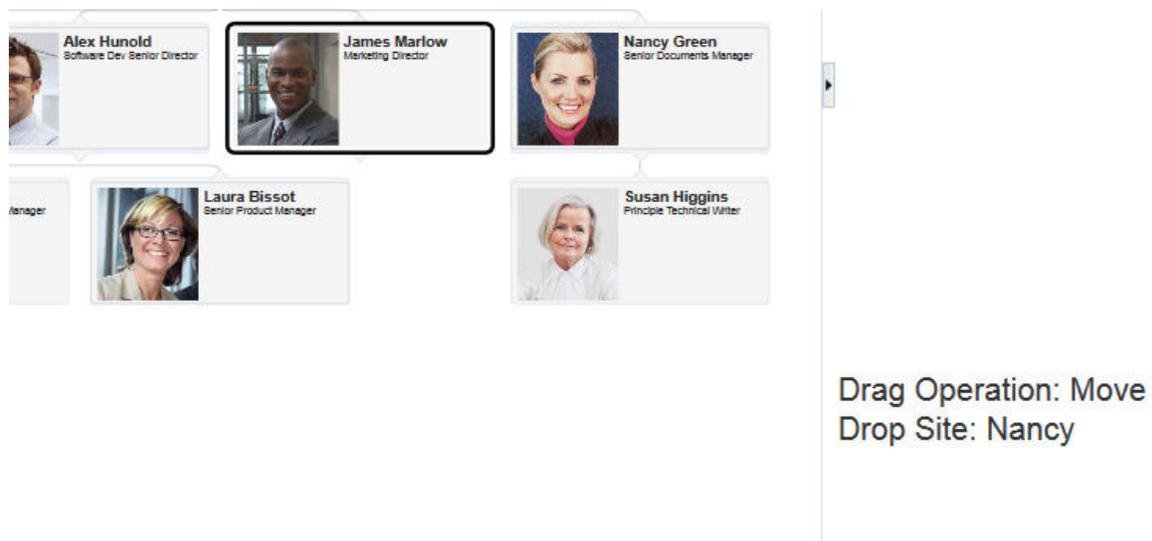
Figure 31-21 shows a portion of the same hierarchy viewer after the user dragged the nodes containing the data for Nina Evans and James Marlow to the drop text.

Figure 31-21 Hierarchy Viewer After Multiple Node Drag



If the user drags from the drag text to a hierarchy viewer node or background, the text will change to indicate where the text was dragged and which operation was performed. Figure 31-22 shows a portion of the same hierarchy viewer after a user drags the text to the hierarchy viewer background.

Figure 31-22 Hierarchy Viewer After Text Drag to Hierarchy Viewer Background



How to Configure Hierarchy Viewer Drag and Drop

To add drag support to a hierarchy viewer, which will allow components or other hierarchy viewers to drag nodes from it, add the `af:dragSource` tag to the hierarchy

viewer and add the `af:dropTarget` tag to the component receiving the drag. The component receiving the drag must include the `org.apache.myfaces.trinidad.model.RowKeySet` data flavor as a child of the `af:dropTarget` and also define a `dropListener` method to respond to the drop event.

To add drop support to a hierarchy viewer, which will allow components or other hierarchy viewers to drag items to it, add the `af:dropTarget` tag to the hierarchy viewer and include the data flavors that the hierarchy viewer will support. Add a `dropListener` method to a managed bean that will respond to the drop event.

The following procedure shows how to set up a hierarchy as a simple drag source or drop target for the `af:outputFormatted` component shown in [Figure 31-20](#). For more detailed information about configuring drag and drop on ADF Faces or ADF Data Visualization components, see [Adding Drag and Drop Functionality](#).

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You will need to complete these tasks:

- Add a hierarchy viewer to your page. For more information, see [How to Add a Hierarchy Viewer to a Page](#).
- Create any additional components needed to support the drag and drop.

For example, the page in [Figure 31-20](#) uses an `af:panelGroupLayout` component containing `af:outputFormatted` and `af:panelList` components to provide instructions to the user. The page also uses an `af:panelSplitter` component to separate the drag and drop `af:outputFormatted` component text from the hierarchy viewer.

The code example below shows the completed page for the additional components. The hierarchy viewer details are omitted.

```
<af:panelStretchLayout id="psl1" topHeight="auto" endWidth="auto">
  <f:facet name="top">
    <af:panelGroupLayout id="pgl2" layout="horizontal">
      <af:spacer width="10px" id="s8"/>
      <af:panelGroupLayout id="pgl14">
        <af:outputFormatted value="Hierarchy Viewer Drag and Drop
Example"
                          id="of4" inlineStyle="font-size:small;
                          font-weight:bold;"/>
        <af:panelList id="pll"
                      inlineStyle="font-size:x-small;"/>
        <af:outputFormatted value="Click and hold on a node for more
than one-half second to initiate the drag. Use Ctrl+Click to select
multiple nodes."
                          id="of1" inlineStyle="font-size:x-small;"/>
        <af:outputFormatted value="Drag one or more nodes from the
hierarchy viewer to the drop text. The text will change to show which
```

```

node(s) you dragged and the operation performed."
        id="of2" inlineStyle="font-size:x-small;"/>
        <af:outputFormatted value="Drag the drag text to one of the
hierarchy viewer nodes or background. The text will change to show
where you dropped it and the operation performed."
        id="of3" inlineStyle="font-size:x-small;"/>
    </af:panelList>
</af:panelGroupLayout>
</af:panelGroupLayout>
</f:facet>
<f:facet name="center">
    <af:panelSplitter id="ps1" orientation="horizontal"
splitterPosition="125"
        positionedFromEnd="false"
styleClass="AFStretchWidth">
    <f:facet name="first">
        <af:panelSplitter id="ps2" orientation="vertical">
            <f:facet name="first">
                <af:panelGroupLayout id="pgl3" layout="vertical">
                    <af:separator id="s1"/>
                    <af:outputFormatted value="#{hvBean.dropText}"
                        clientComponent="true"
                        inlineStyle="font-size:small; font-weight:bold;"
                        id="of5">
                        <af:dropTarget actions="COPY MOVE LINK"

dropListener="#{hvBean.fromDropListener}">
                            <af:dataFlavor

flavorClass="org.apache.myfaces.trinidad.model.RowKeySet"/>
                                <af:dropTarget>
                                    </af:outputFormatted>
                                    <af:spacer width="200" id="s11"/>
                                    <af:separator id="s3"/>
                                </af:panelGroupLayout>
                                </f:facet>
                                <f:facet name="second">
                                    <af:panelGroupLayout id="pgl1">
                                        <af:separator id="s12"/>
                                        <af:outputFormatted value="#{hvBean.dragText}"
                                            clientComponent="true"
                                            inlineStyle="font-size:small; font-weight:bold;"
                                            id="of6">
                                            <af:componentDragSource/>
                                                </af:outputFormatted>
                                                <af:separator id="s10"/>
                                            </af:panelGroupLayout>
                                        </f:facet>
                                    </af:panelSplitter>
                                </f:facet>
                                <f:facet name="second">
                                    <dvt:hierarchyViewer id="shv" styleClass="AFStretchWidth"
                                        controlPanelBehavior="initExpanded"
                                        var="node" detailWindow="none"
                                        value="#{XMLParser.employees}"

```

```

        contentDelivery="immediate"
        summary="HV Drag and Drop Sample"

navigateUpListener="#{XMLParser.doNavigateUp}">
    <af:dragSource actions="COPY MOVE LINK"
defaultAction="MOVE"/>
    <af:dropTarget actions="COPY MOVE LINK"
        dropListener="#{hvBean.toDropListener}">
        <af:dataFlavor flavorClass="java.lang.Object"/>
    </af:dropTarget>
    <dvt:link linkType="orthogonalRounded" id="l1"/>
    <dvt:node width="233" height="330"
        setAnchorListener="#{XMLParser.doSetAnchor}" id="n1"
        showNavigateUp="#{node.topNode == false}"
        showExpandChildren="#{node.hasChildren}">
        <f:facet name="zoom100">
            < remaining hierarchy viewer contents omitted >
        </dvt:hierarchyViewer>
    </f:facet>
    </af:panelSplitter>
</f:facet>
</af:panelStretchLayout>

```

For additional information about `af:outputFormatted` components, see [Using Output Components](#). For help with the `af:panelGroupLayout` component or other page layout components, see [Organizing Content on Web Pages](#).

To configure hierarchy viewer drag and drop:

1. To configure a hierarchy viewer as a drop target, in the Components window, from the Operations panel, drag a **Drop Target** and drop it as a child to the hierarchy viewer.
2. In the Insert Drop Target dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the hierarchy viewer's managed bean. Alternatively, use the dropdown menu to choose **Expression Builder** and enter an EL Expression for the drop listener.

For example, to add a method named `toDropListener()` on a managed bean named `hvBean`, choose **Edit**, select **hvBean** from the dropdown menu, and click **New** on the right of the **Method** field to create the `toDropListener()` method.

The example below shows the sample drop listener and supporting methods for the hierarchy viewer displayed in [Figure 31-20](#).

```

// imports needed by methods
import java.util.Map;
import oracle.adf.view.rich.dnd.DnDAction;
import oracle.adf.view.rich.event.DropEvent;
import oracle.adf.view.rich.datatransfer.DataFlavor;
import oracle.adf.view.rich.datatransfer.Transferable;
import org.apache.myfaces.trinidad.context.RequestContext;
import org.apache.myfaces.trinidad.render.ClientRowKeyManager;
import javax.faces.context.FacesContext;
import
oracle.adf.view.faces.bi.component.hierarchyViewer.UIHierarchyViewer;
import javax.faces.component.UIComponent;

```

```

// variables need by methods
private String dragText = "Drag this text onto a node or the hierarchy
viewer background";
// drop listener
public DnDAction toDropListener(DropEvent event) {
    Transferable transferable = event.getTransferable();
    DataFlavor<Object> dataFlavor =
DataFlavor.getDataFlavor(Object.class);
    Object transferableObj = transferable.getData(dataFlavor);
    if(transferableObj == null)
        return DnDAction.NONE;
    // Build up the string that reports the drop information
    StringBuilder sb = new StringBuilder();
    // Start with the proposed action
    sb.append("Drag Operation: ");
    DnDAction proposedAction = event.getProposedAction();
    if(proposedAction == DnDAction.COPY) {
        sb.append("Copy<br>");
    }
    else if(proposedAction == DnDAction.LINK) {
        sb.append("Link<br>");
    }
    else if(proposedAction == DnDAction.MOVE) {
        sb.append("Move<br>");
    }
    // Then add the rowKeys of the nodes that were dragged
    UIComponent dropComponent = event.getDropComponent();
    Object dropSite = event.getDropSite();
    if(dropSite instanceof Map) {
        String clientRowKey = (String) ((Map) dropSite).get("clientRowKey");
        Object rowKey = getRowKey(dropComponent, clientRowKey);
        sb.append("Drop Site: ");
        if(rowKey != null) {
            sb.append("Node: ");
            sb.append(getLabel(dropComponent, rowKey));
        }
        else {
            sb.append("Background");
        }
    }
    // Update the output text
    this._dragText = sb.toString();

RequestContext.getCurrentInstance().addPartialTarget(event.getDragCompon
ent());
    return event.getProposedAction();
}
private String getLabel(UIComponent component, Object rowKey) {
    if(component instanceof UIHierarchyViewer) {
        UIHierarchyViewer hv = (UIHierarchyViewer) component;
        Employee rowData = (Employee) hv.getRowData(rowKey);
        return rowData.getFirstName() + " " + rowData.getLastName();
    }
    return null;
}
}

```

```

private Object getRowKey(UIComponent component, String clientRowKey) {
    if(component instanceof UIHierarchyViewer) {
        UIHierarchyViewer hv = (UIHierarchyViewer) component;
        ClientRowKeyManager crkm = hv.getClientRowKeyManager();
        return crkm.getRowKey(FacesContext.getCurrentInstance(), component,
clientRowKey);
    }
    return null;
}
public String getDragText() {
    return _dragText;
}

```

 **Note:**

This method references an `Employee` class that defines the attributes for the hierarchy viewer. If your hierarchy viewer uses a different class, substitute the name of that class instead.

If you want to look at the source code for the `Employee` class used in this example, you can find the source code for it and other supporting classes in the ADF Faces Components Demo application. For more information about the demo application, see [ADF Faces Components Demo Application](#).

3. Click **OK** to enter the Insert Data Flavor dialog.
4. In the Insert Data Flavor dialog, enter the object that the drop target will accept. Alternatively, use the dropdown menu to navigate through the object hierarchies and choose the desired object.

For example, to allow the `af:outputFormatted` component to drag text to the hierarchy viewer, enter `java.lang.Object` in the Insert Data Flavor dialog.

5. In the Structure window, right-click the **af:dropTarget** node and choose **Go to Properties**.
6. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

7. To use the hierarchy viewer as the drop target, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the component that will be the source of the drag.

For example, drag and drop a **Drag Source** as a child to an `af:outputFormatted` component.
 - b. In the Structure window, right-click the **af:dragSource** node and choose **Go to Properties**.
 - c. In the component's **Value** field, reference the public variable that you created in the drop listener for the hierarchy viewer in Step 2.

For example, for a drop listener named `toDropListener()` and a variable named `dropText`, enter the following in the component's **Value** field:

```
#{hvBean.dropText}
```

8. To configure the hierarchy viewer as a drag source, in the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the hierarchy viewer.
9. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

10. To specify the default action that the drag source will support, use the **DefaultAction** attribute's dropdown menu to choose `COPY`, `MOVE`, or `LINK`.

The hierarchy viewer in the drag and drop example in [Figure 31-20](#) uses `MOVE` as the default action.

11. To make another component the drop target for drags from the hierarchy viewer, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drop Target** onto the component that will receive the drop.

For example, the page in the drag and drop example in [Figure 31-20](#) contains an `af:outputFormatted` component that displays the results of the drop.

- b. In the Insert Drop Target dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the appropriate managed bean. Alternatively, use the dropdown menu to choose **Expression Builder** and enter an EL Expression for the drop listener.

For example, to add a method named `fromDropListener()` on a managed bean named `hvBean`, choose **Edit**, select `hvBean` from the dropdown menu, and click **New** on the right of the **Method** field to create the `fromDropListener()` method.

The example below shows the sample drop listener for the hierarchy viewer displayed in [Figure 31-20](#). This example uses the same imports and helper methods used in, and they are not included here.

```
// Additional import needed for listener
import org.apache.myfaces.trinidad.model.RowKeySet;
// Variables needed by method
private String dropText = "Drop a node here";
// Drop listener
public DnDAction fromDropListener(DropEvent event) {
    Transferable transferable = event.getTransferable();
    DataFlavor<RowKeySet> dataFlavor =
DataFlavor.getDataFlavor(RowKeySet.class);
    RowKeySet rowKeySet = transferable.getData(dataFlavor);
    if(rowKeySet == null || rowKeySet.getSize() <= 0)
        return DnDAction.NONE;
    // Build up the string that reports the drop information
    StringBuilder sb = new StringBuilder();
    // Start with the proposed action
```

```

sb.append("Drag Operation: ");
DnDAction proposedAction = event.getProposedAction();
if(proposedAction == DnDAction.COPY) {
    sb.append("Copy<br>");
}
else if(proposedAction == DnDAction.LINK) {
    sb.append("Link<br>");
}
else if(proposedAction == DnDAction.MOVE) {
    sb.append("Move<br>");
}
// Then add the rowKeys of the nodes that were dragged
sb.append("Nodes: ");
UIComponent dragComponent = event.getDragComponent();
for(Object rowKey : rowKeySet) {
    sb.append(getLabel(dragComponent, rowKey));
    sb.append(", ");
}
// Remove the trailing ,
sb.setLength(sb.length()-2);
// Update the output text
this.dropText = sb.toString();

RequestContext.getCurrentInstance().addPartialTarget(event.getDropComponent());
return event.getProposedAction();
}

```

- c. Click **OK** to enter the Insert Data Flavor dialog.
- d. In the Insert Data Flavor dialog, enter `org.apache.myfaces.trinidad.model.RowKeySet`.
For example, to allow the `af:outputFormatted` component to drag text to the hierarchy viewer, enter `org.apache.myfaces.trinidad.model.RowKeySet` in the Insert Data Flavor dialog.
- e. In the Structure window, right-click the **af:dropTarget** node and choose **Go to Properties**.
- f. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

- g. In the component's value field, reference the public variable that you created in the drop listener for the treemap or sunburst in Step 2.

For example, for a drop listener named `fromDropListener()` and a variable named `dragText`, enter the following in the component's **Value** field:

```
#{hvBean.dragText}
```

What You May Need to Know About Configuring Hierarchy Viewer Drag and Drop

You can disable the ability to drag a node by setting its `draggable` attribute to `no`.

Adding Search to a Hierarchy Viewer

The ADF DVT hierarchy viewer component search functionality looks through the data structure of the hierarchy viewer and presents matches in a scrollable list. Users can double-click a search result to display the matching node as the anchor node in the hierarchy viewer. When enabled, a search panel is displayed in the upper right-hand corner of the hierarchy viewer, and results are displayed below the search panel.

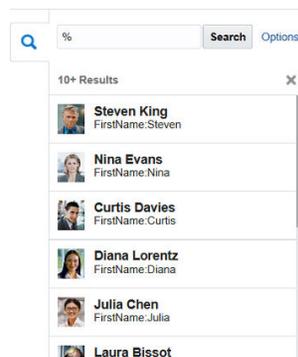
Figure 31-23 shows a sample search panel.

Figure 31-23 Hierarchy Viewer Search Panel



Figure 31-24 shows sample search results.

Figure 31-24 Hierarchy Viewer Sample Search Results



How to Configure Searching in a Hierarchy Viewer

Add the `dvt:search` tag as a child of the `dvt:hierarchyViewer` tag to enable searching, and `dvt:searchResults` as a child of `dvt:search` to specify how to handle the results.

Search in a hierarchy viewer is based on the searchable attributes or columns of the data collection that is the basis of the hierarchy viewer data model. Using a query results collection defined in data controls in Oracle ADF, JDeveloper makes this a declarative task. For more information, see the "How to Create a Databound Search in a Hierarchy Viewer" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see [Configuring Hierarchy Viewer Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Hierarchy Viewer Components](#).

You should already have a hierarchy viewer on your page. If you do not, follow the instructions in this chapter to create a hierarchy viewer. For more information, see [How to Add a Hierarchy Viewer to a Page](#).

To configure search in a hierarchy viewer:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Insert Inside Hierarchy Viewer > Search**.
2. In the Properties window, set the following attributes to configure the search functionality:
 - **Value:** Specify the variable to hold the search text.
 - **ActionListener:** Enter the listener called to perform the search.
 - **InitialBehavior:** Specify how the search panel is initially displayed. Valid values are `initCollapsed` for initially collapsed, `initExpanded` for initially expanded, or `hidden` for completely hidden from view.
3. Optionally, to configure a component that will use the `af:query` component to launch an advanced search outside of the hierarchy viewer component:
 - a. In the Structure window, expand the **dvt:search** node.
 - b. Right-click the **af:link** node and choose **Go to Properties**.
 - c. In the Properties window, configure any desired properties for the **af:link** node.

For example, to change the text that displays *Advanced* for the advanced search, enter a value for the **Text** attribute.
 - d. To add a popup that will display the advanced search options to the user, in the Structure window, right-click the **af:link** node and choose **Insert Inside Link > ADF Faces > Show Popup Behavior**.
 - e. To complete the popup configuration, add the `af:popup` component to your page and configure the `af:query` component to perform the search.

For help with configuring popups, see [Declaratively Invoking a Popup](#). For more information about the `query` component, see [Using the query Component](#).
4. In the Structure window, right-click the **dvt:search** node and choose **Insert Inside Search > Search Result**.
5. In the Properties window, set the following attributes to configure the display of the search results:
 - **Value:** Specify the search results data model. This must be an instance of `oracle.adf.view.faces.bi.model.DataModel`.

- **Var:** Enter the name of the EL variable used to reference each element of the hierarchy viewer collection. Once this component has completed rendering, this variable is removed, or reverted back, to its previous value.
 - **VarStatus:** Enter the name of the EL variable used to reference the `varStatus` information. Once this component has completed rendering, this variable is removed, or reverted back, to its previous value.
 - **ResultListener:** Specify a reference to an action listener that will be called after a row in the search results is selected.
 - **EmptyText:** Specify the text to display when no results are returned.
 - **FetchSize:** Specify the number of result rows to fetch at a time.
6. In the Structure window, right-click the **dvt:searchResults** node and choose **Inside Search Result > Set Property Listener**.
 7. In the Properties window, set the following attributes to map the search results node from the results model to the corresponding hierarchy viewer model:
 - **From:** Specify the source of the value, a constant or an EL expression.
 - **To:** Specify the target of the value.
 - **Type:** Choose `action` as the value.
 8. In the Structure window, do the following to specify the components to stamp out the search results:
 - To wrap the output of the search results, right-click the **f:facet-content** node and choose **Insert Inside Facet > ADF Faces > Panel Group Layout**.
 - To display the search results, insert the ADF Faces output components inside the **af:panelGroupLayout** node to display the search results.

For example, to display output text, right-click the **af:panelGroupLayout** node and choose **Insert Inside Panel Group Layout > Output Text**.

The following output appears in the code after inserting and configuring two `af:outputText` elements:

```
<af:outputText value="#{resultRow.Lastname}" id="ot1"
               inlineStyle="color:blue;"/>
<af:outputText value="#{resultRow.Firstname}" id="ot2"/>
```

Each stamped row references the current row using the `var` attribute of the `dvt:searchResults` tag.

The example below shows sample code for configuring search in a hierarchy viewer.

```
<dvt:hierarchyViewer>
  ... hierarchy viewer details omitted
  <dvt:search id="searchId" value="#{bindings.lastNameParam.inputValue}"
             actionListener="#{bindings.ExecuteWithParams1.execute}">
    <f:facet name="end">
      <af:link text="Advanced" styleClass="AFHVAdvancedSearchLinkStyle"
              id="l2">
        <af:showPopupBehavior popupId=":mypop" triggerType="action"/>
      </af:link>
    </f:facet>
    <dvt:searchResults id="searchResultId"
                     emptyText="#{bindings.searchResult1.viewable ?
```

```
'No match.' : 'Access Denied.}'"
    fetchSize="25"
    value="#{bindings.searchResult1.collectionModel}"

resultListener="#{bindings.ExecuteWithParams.execute}"
    var="resultRow">
    <af:setPropertyListener from="#{resultRow.Id}"
        to="#{bindings.employeeId.inputValue}"
        type="action"/>
    <f:facet name="content">
    <af:panelGroupLayout layout="horizontal">
    <af:outputText value="#{resultRow.Lastname}" id="ot1"
        inlineStyle="color:blue;"/>
    <af:outputText value="#{resultRow.Firstname}" id="ot2"/>
    </af:panelGroupLayout>
    </f:facet>
    </dvt:searchResults>
</dvt:search>
</dvt:hierarchyViewer>
```

What You May Need to Know About Configuring Search in a Hierarchy Viewer

The search results stamp display in the size specified in its outermost container, the `af:panelGroupLayout`. By default, this size is 100 x 30 pixels. If you need to adjust the size of the search results display, configure the `InlineStyle` attribute of the `af:panelGroupLayout`.

For more information, see [How to Use the `panelGroupLayout` Component](#).

Using Treemap and Sunburst Components

This chapter describes how to use the ADF Data Visualization `treemap` and `sunburst` components to display hierarchical data in treemaps and sunbursts using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the components.

If your application uses the Fusion technology stack, then you can also use data controls to create treemaps and sunbursts. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Treemap and Sunburst Components](#)
- [Using the Treemap and Sunburst Components](#)
- [Adding Data to Treemap and Sunburst Components](#)
- [Customizing Treemap and Sunburst Display Elements](#)
- [Adding Interactive Features to Treemaps and Sunbursts](#)

About the Treemap and Sunburst Components

ADF DVT Treemaps and Sunbursts are visualizations to provide a two-dimensional, aesthetically pleasing representation of hierarchical data. Each data component is represented using a shape called a node. Use Treemaps and Sunbursts to easily identify significant data and points of interest.

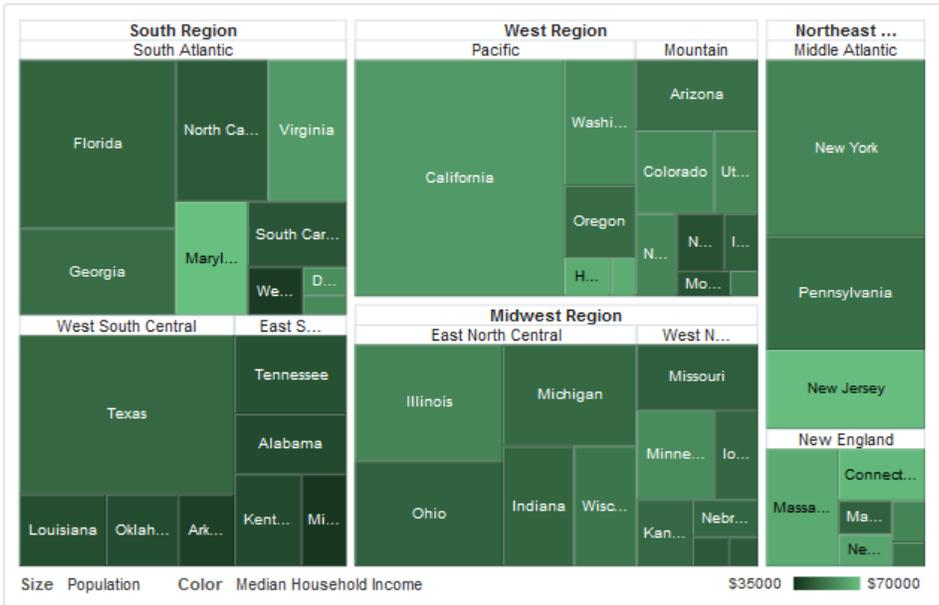
For example, you can use a treemap or sunburst to display quarterly regional sales and to identify sales trends, using the size of the node to indicate each region's sales volume and the node's color to indicate whether that region's sales increased or decreased over the quarter. The appearance and content of the nodes are configurable at each level of the hierarchy.

Treemap and Sunburst Use Cases and Examples

Treemaps display nodes as a set of nested rectangles. Each branch of the tree is given a rectangle, which is then tiled with smaller rectangles representing sub-branches.

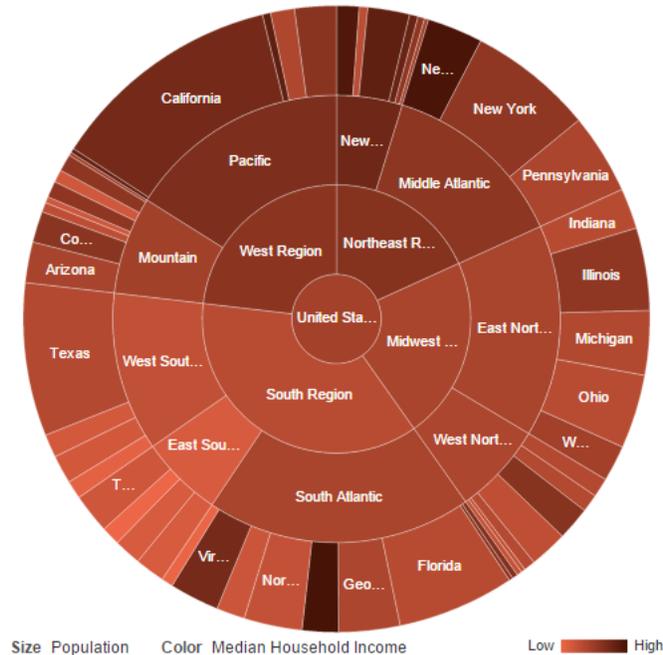
[Figure 32-1](#) shows a treemap displaying United States census data grouped by regions, with the color attribute used to indicate median income levels. States with larger populations display in larger-sized nodes than states with smaller populations.

Figure 32-1 Treemap Displaying United States Population and Median Income by Regions



Sunbursts display the nodes in a radial rather than a rectangular layout, with the top of the hierarchy at the center and deeper levels farther away from the center. [Figure 32-2](#) shows the same census data displayed in a sunburst.

Figure 32-2 Sunburst Displaying United States Population and Median Income by Regions



Both treemaps and sunbursts can display thousands of data points in a relatively small spatial area. These components are a good choice for identifying trends for large hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information.

Use treemaps if you are primarily interested in displaying two metrics of data using size and color at a single layer of the hierarchy. Use sunbursts instead if you want to display the metrics for all levels in the hierarchy. Drilling can be enabled to allow the end user to traverse the hierarchy and focus in on key parts of the data.

If your application uses a smaller data set or you wish to emphasize the parent/child relationship between the nodes, then consider using the `tree`, `treeTable`, or `hierarchyViewer` component. For information about using trees and tree tables, see [Using Tables, Trees, and Other Collection-Based Components](#). For information about using hierarchy viewers, see [Using Hierarchy Viewer Components](#).

End User and Presentation Features of Treemaps and Sunbursts

The ADF Data Visualization `treemap` and `sunburst` components provide a range of features for end users, such as drilling, grouping, and filtering. They also provide a range of presentation features, such as layout variations, legend display, and customizable colors and label styles.

To use and customize treemap and sunburst components, it may be helpful to understand these features and components:

Treemap and Sunburst Layouts

You define the initial layout of the treemap or sunburst when you insert the component on the page from either the Data Controls panel to bind a data collection to the `treemap` or `sunburst` component or from the Components window to insert the component.

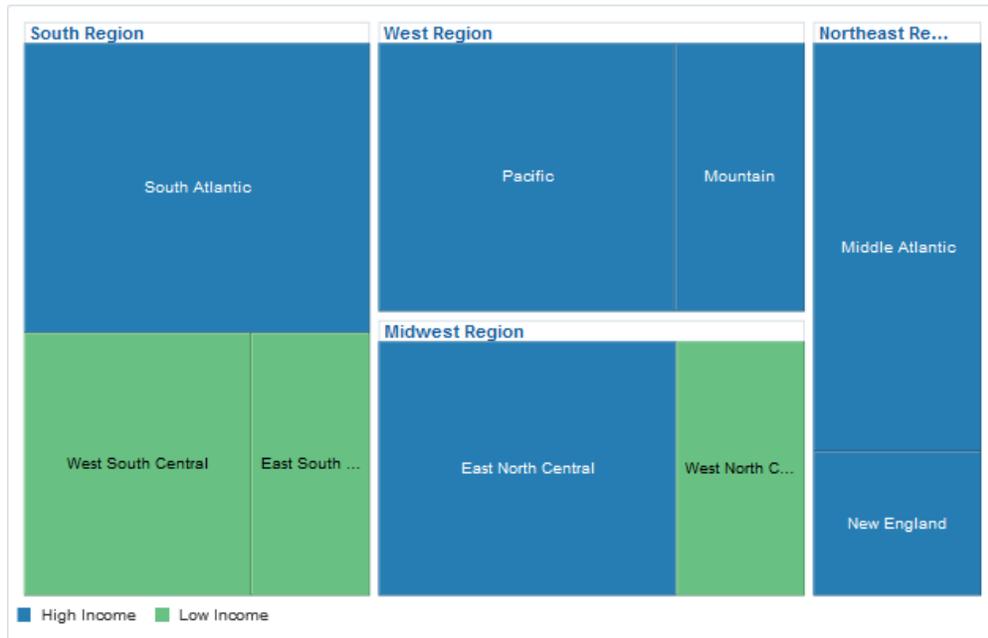
The `sunburst` component has only one layout, as shown in [Figure 32-2](#).

The layout of nodes in a `treemap` component is configurable and includes the following types of layouts:

- Squarified, nodes are laid out to be as square as possible.

The squarified layout is optimized so that the user can most easily compare the relative sizes of the nodes and is the layout displayed in [Figure 32-3](#).

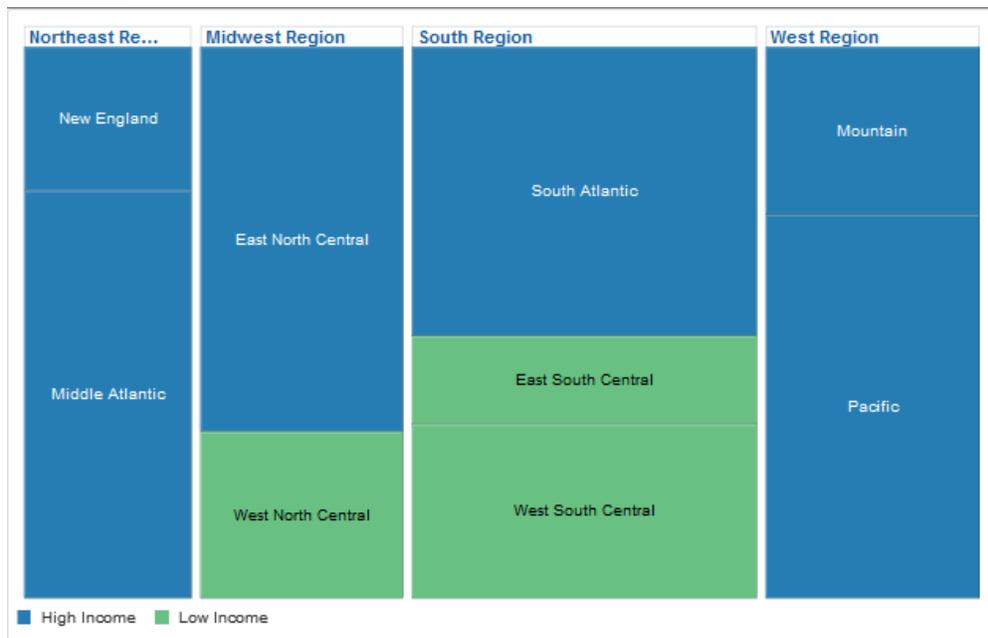
Figure 32-3 Treemap Displaying United States Data in Squarified Layout



- Slice and dice horizontal, nodes are first laid out horizontally across the width of the treemap and then vertically across the height of the treemap.

This layout is optimized for animation because the relative ordering of the nodes remains constant. [Figure 32-4](#) displays the sample United States census data in the slice and dice horizontal layout.

Figure 32-4 Treemap Displaying United States Census Data in Slice and Dice Horizontal Layout



- Slice and dice vertical, nodes are first laid out vertically across the height of the treemap and then horizontally across the width of the treemap.

This layout is also optimized for animation because the relative ordering of the nodes remains constant. [Figure 32-5](#) displays the sample United States census data in the slice and dice vertical layout.

Figure 32-5 Treemap Displaying United States Census Data in Slice and Dice Vertical Layout

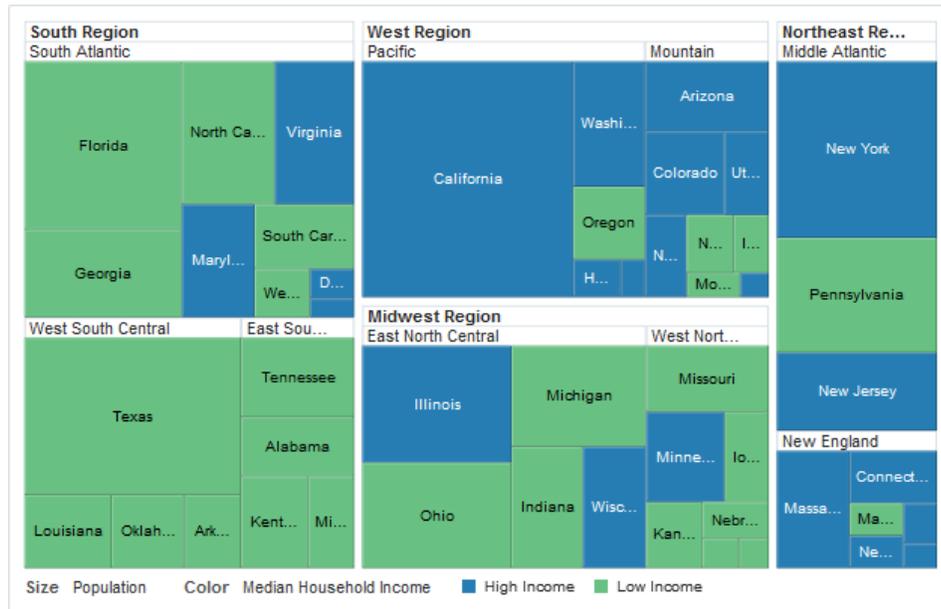


Attribute Groups

Treemaps and sunbursts support the use of the `dvt:attributeGroups` tag to generate stylistic attribute values such as colors for each unique group in the data set.

In treemaps and sunbursts, the data values determine which color to display. Both components display continuous data by selecting a color across a gradient, in which the nodes change color gradually based on the data values. The treemap in [Figure 32-1](#) uses gradients to display the median income as a range of data.

For discrete data, treemaps and sunbursts display a specific color, also based on the data value. [Figure 32-6](#) shows the same United States census population using two colors to distinguish between high and low median incomes.

Figure 32-6 Treemap Displaying Discrete Attribute Groups

Attribute groups are required for legend support and make it easier to customize the color and pattern of the nodes.

Legend Support

Treemaps and sunbursts display legends below the components to provide a visual clue to the type of data controlling the size and color. If the component uses attribute groups to specify colors based on conditions such as income levels, the legend can also display the colors used and indicate what value each color represents.

The treemap in [Figure 32-6](#) displays a legend for a treemap using discrete attribute groups. The legend makes it easy to determine which colors are used to indicate low or high median incomes.

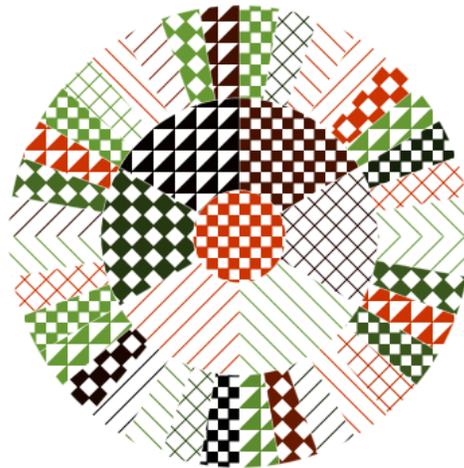
If your treemap uses continuous attribute groups, the legend displays the colors used as a gradient. The treemap in [Figure 32-1](#) shows a legend for a treemap using continuous attribute groups to indicate median income levels.

Pattern Support

Treemaps and sunbursts display patterns when values are specified for the `fillPattern` attribute on the child nodes. The pattern is drawn with a white background, and the `fillColor` value determines the foreground color.

[Figure 32-7](#) shows a sunburst configured with an assortment of fill patterns.

Figure 32-7 Sunburst Illustrating Pattern Fill

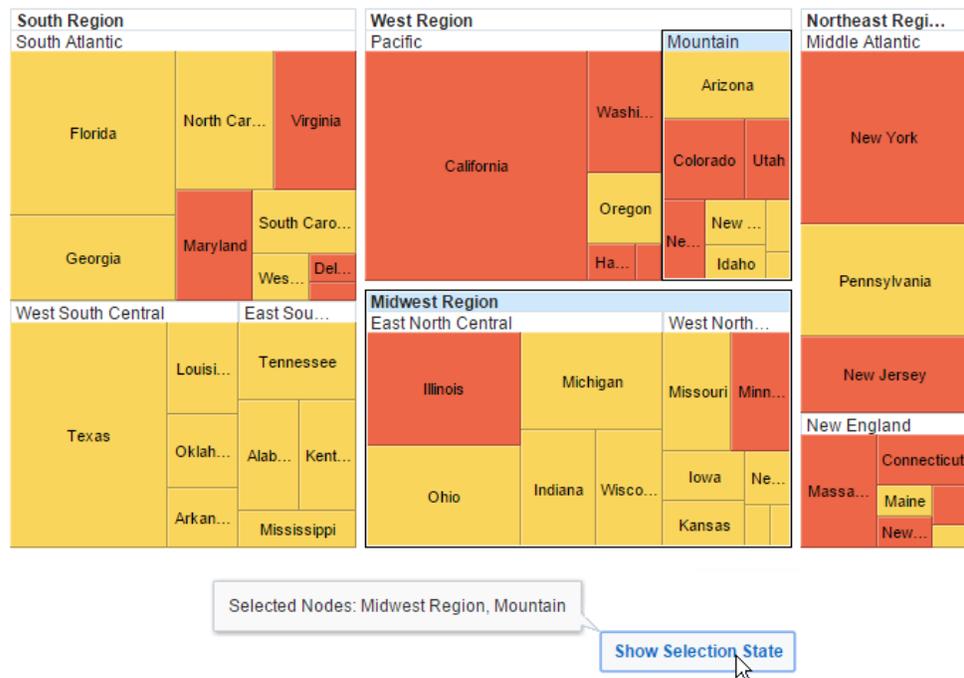


Node Selection Support

Treemaps and sunbursts support the ability to respond to user clicks on one or more nodes to display information about the selected node(s).

Figure 32-8 shows a treemap configured for multiple selection support.

Figure 32-8 Treemap Illustrating Multiple Selection Support

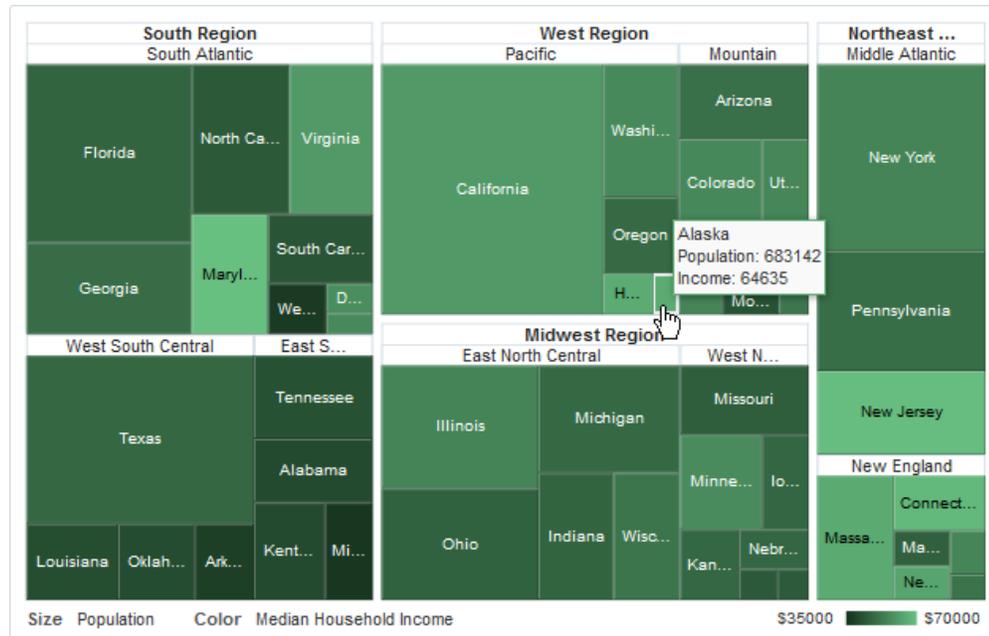


Tooltip Support

Treemaps and sunbursts support the ability to display additional information about a node when the user moves the mouse over a node.

Figure 32-9 shows the tooltip that is displayed when the user moves the mouse over the Alaska node.

Figure 32-9 Treemap Displaying a Tooltip



The tooltip permits the user to see information about the data that may not be obvious from the visual display. Configuring tooltips on treemaps and sunbursts is recommended due to the space-constrained nature of the components.

Popup Support

Treemap and sunburst components can be configured to display popup dialogs, windows, and menus that provide information or request input when the user clicks or hovers the mouse over a node.

Figure 32-10 shows a sample popup displayed when a user hovers the mouse over one of the treemap nodes.

Figure 32-10 Treemap Popup on Mouse Hover

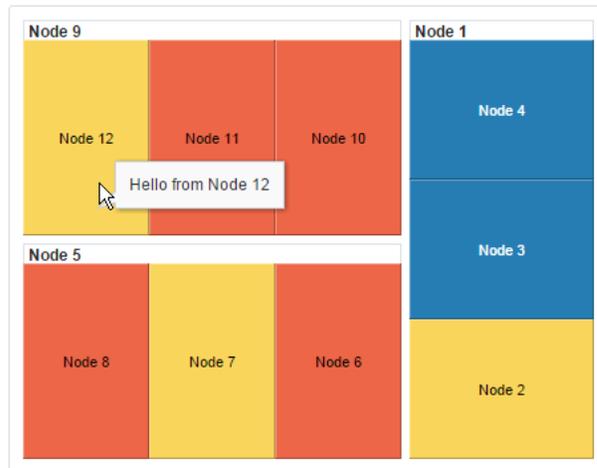
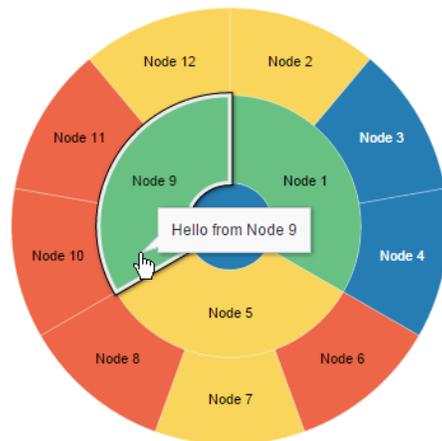


Figure 32-11 shows a similar popup window that is displayed when the user clicks on one of the sunburst nodes.

Figure 32-11 Sunburst Popup on Mouse Click

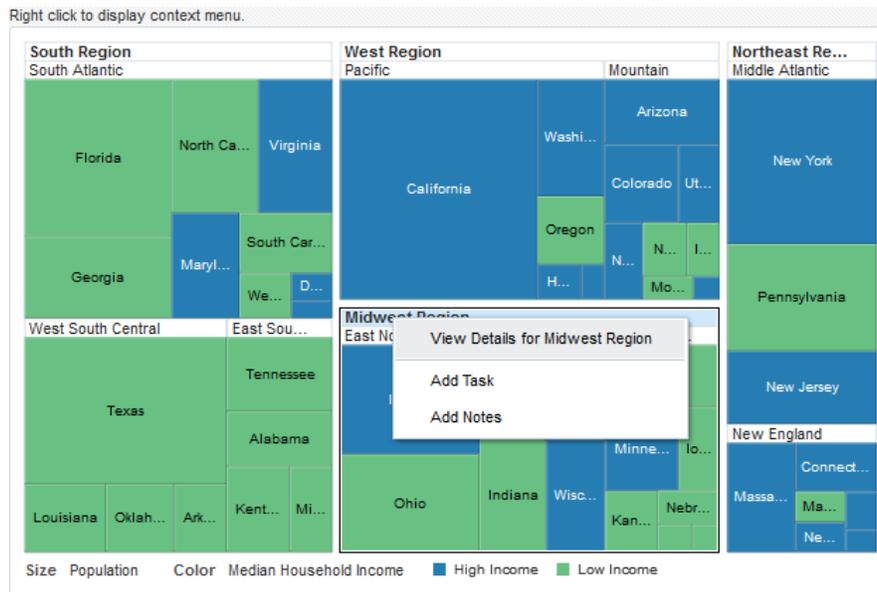


Context Menus

Treemaps and sunbursts support the ability to display context menus to provide additional information about the selected node.

Figure 32-12 shows a context menu displayed when the user right-clicks on one of the sunburst nodes.

Figure 32-12 Treemap Context Menu

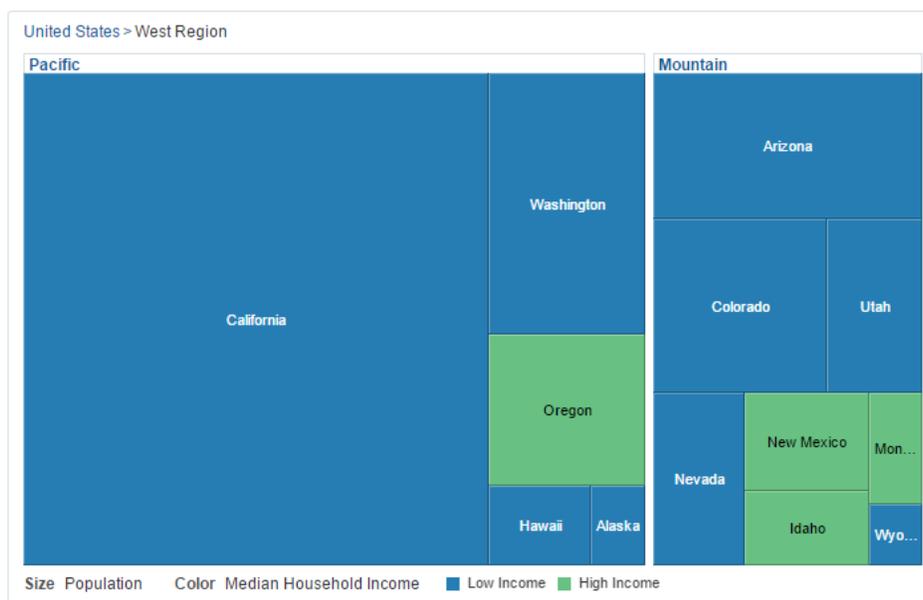


Drilling Support

Treemap and sunburst components support drilling to navigate through the hierarchy and display more detailed information about a node.

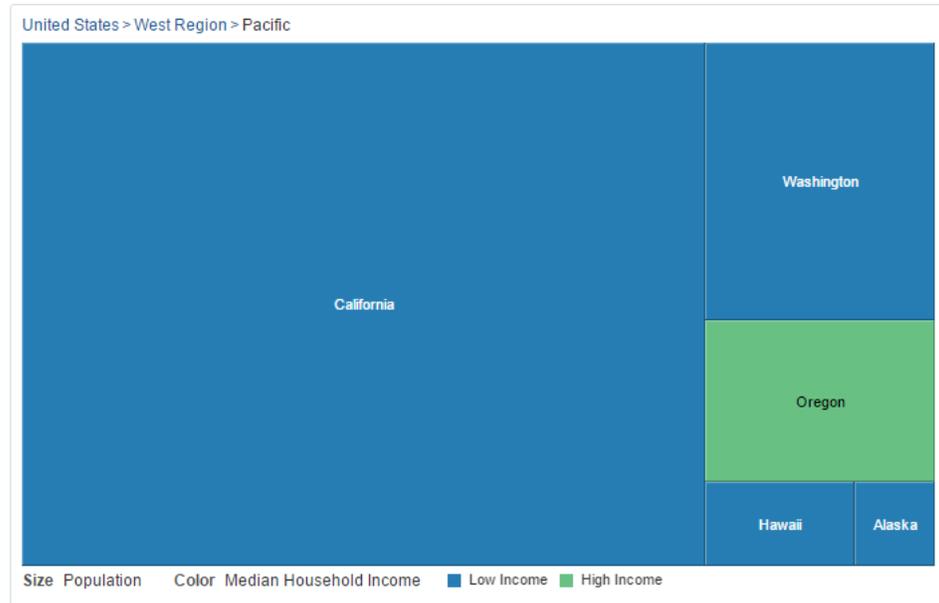
Figure 32-13 shows the treemap that is displayed when a user clicks on the West Region header text in Figure 32-6. The user can navigate back up the hierarchy by clicking on the **United States > West Region** breadcrumb.

Figure 32-13 Treemap Drilled on a Group Node Header



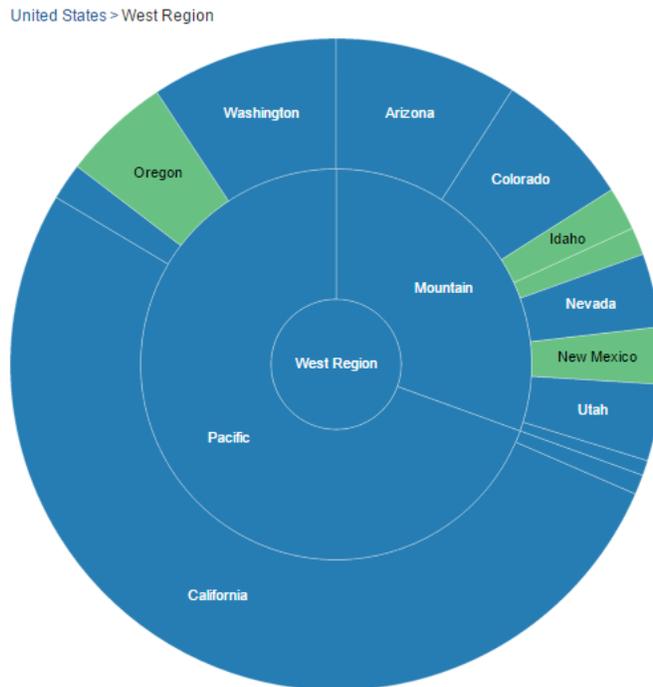
The user can also double-click on a node to set the node as the root of the hierarchy as shown in [Figure 32-14](#).

Figure 32-14 Treemap Drilled on a Node



To drill on a `sunburst` component, the user double-clicks a sunburst node to set it as the root of the hierarchy as shown in [Figure 32-15](#). The user can navigate back up the hierarchy by clicking the **United States > West Region** breadcrumb or by pressing the shift key and double-clicking the **West Region** node.

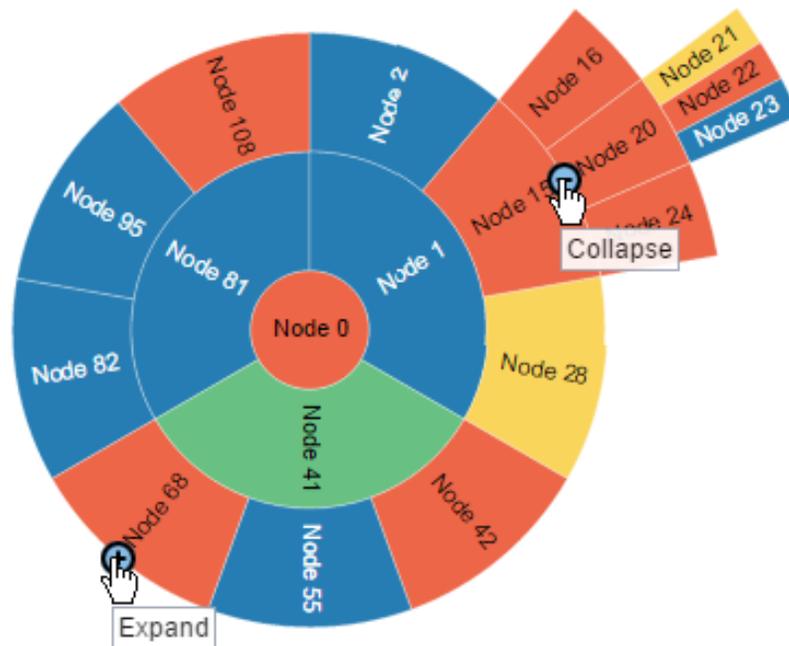
Figure 32-15 Sunburst Drilled on a Node



Sunbursts also provide the ability to expand or collapse the children of a selected node. Users click the **Expand** icon that appears when the user moves the mouse over the node to expand it. To collapse the children, users click the **Collapse** icon.

[Figure 32-16](#) shows a sunburst configured for asymmetric drilling.

Figure 32-16 Sunburst Configured for Asymmetric Drilling

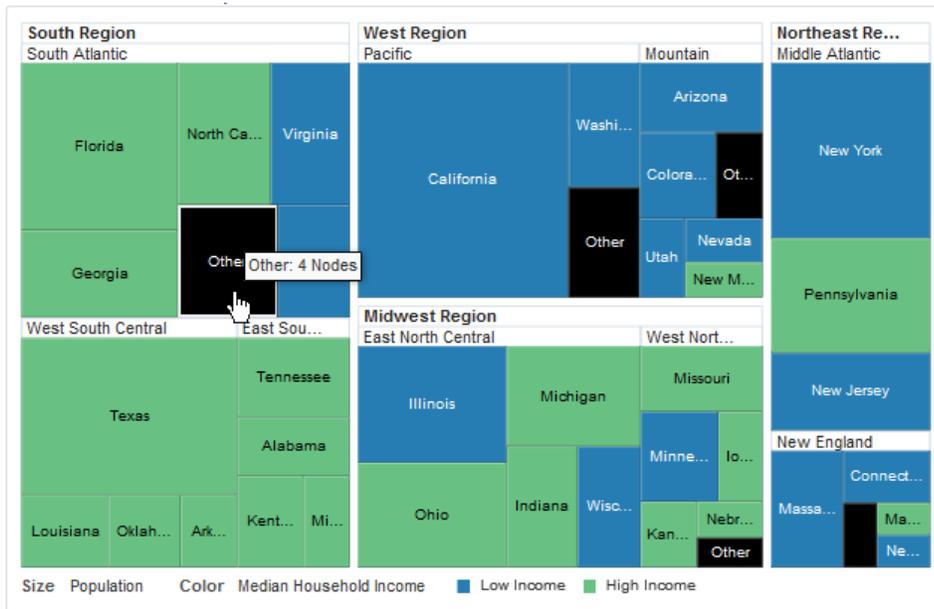


Other Node Support

Treemap and sunburst components provide the ability to aggregate data if your data model includes a large number of smaller contributors in relation to the larger contributors.

Figure 32-17 shows the census treemap displayed in Figure 32-6 with the **Other** node configured. In this example, the **South Carolina, Delaware, West Virginia, and District of Columbia** nodes in the South Atlantic region are represented by the **Other** node.

Figure 32-17 Treemap Displaying Other Node



Drag and Drop Support

Treemap and sunburst components support drag and drop both as a drop source and a drop target.

Figure 32-18 shows a treemap configured as a drag source. When the user drags one of the nodes to the text on the right, the text changes to reflect which node was dragged.

Figure 32-18 Treemap Configured as a Drag Source

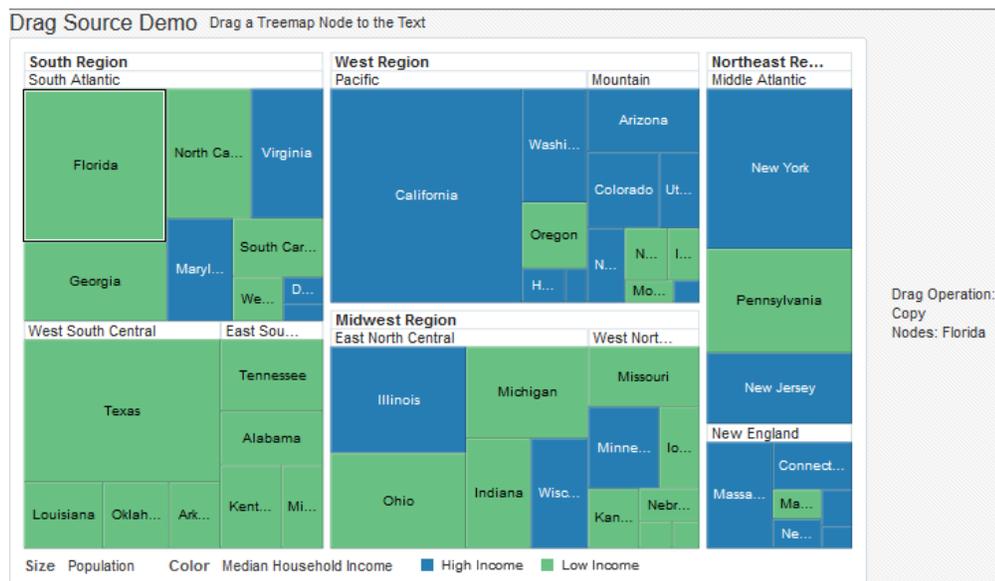
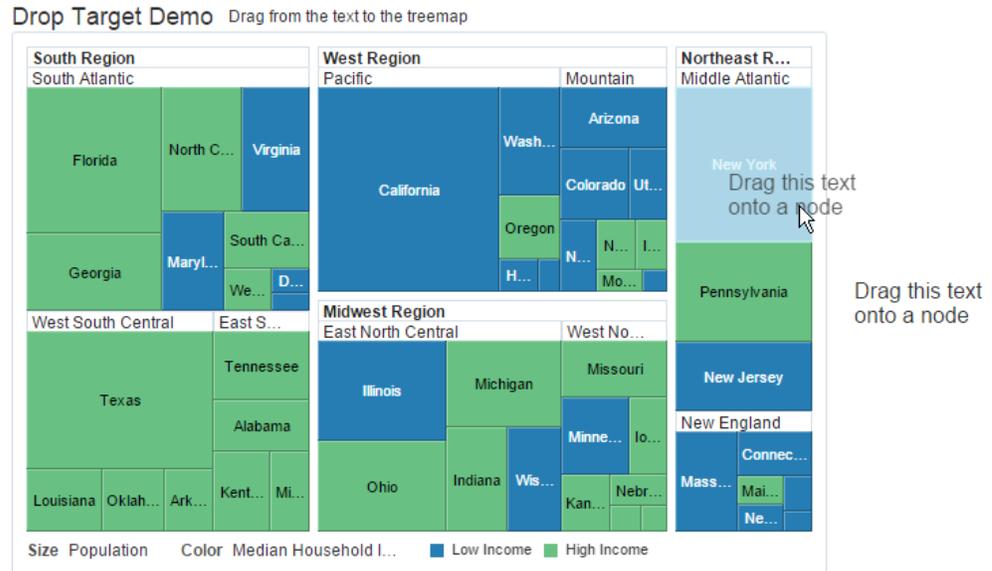


Figure 32-19 shows a treemap configured as a drop target. When the user drags the text on the right to one of the nodes, the text changes to reflect which node received the text.

Figure 32-19 Treemap Configured as a Drop Target



Sorting Support

Treemap and sunburst components support sorting to display nodes with the same parent by size. This feature is useful if your data model is not already sorted because it makes comparison of the nodes easier.

Figure 32-20 shows a sorted treemap. The nodes are arranged in decreasing size, making it easy to see which regions have the largest population.

Figure 32-20 Sorted Treemap





Note:

Treemaps support sorting in the slice and dice layouts only.

Treemap and Sunburst Image Formats

Treemaps and sunbursts support the following image formats: HTML5, Flash, and Portable Network Graphics (PNG).

By default, treemaps and sunbursts will display in the best output format supported by the client browser. If the best output format is not available on the client, the application will default to an available format. For example, if the client does not support HTML5, the application will use:

- Flash, if the Flash Player is available.
You can control the use of Flash content across the entire application by setting a `flash-player-usage` context parameter in `adf-config.xml`. For more information, see [Configuring Flash as Component Output Format](#).
- PNG output format. Although static rendering is fully supported when using a PNG output format, certain interactive features are not available including:
 - Animation
 - Context menus
 - Drag and drop gestures
 - Popup support
 - Selection

Advanced Node Content

Treemaps and sunbursts provide a content facet on the nodes to add content that would not normally fit into a text label. For sunbursts, the advanced content is displayed on the root node. For treemaps, the advanced content is displayed on the leaf nodes.

[Figure 32-21](#) shows an example of a sunburst using advanced node content on the root node. In this example, the root node displays an image and title in addition to the node text.

Figure 32-21 Sunburst Displaying Advanced Root Node Content

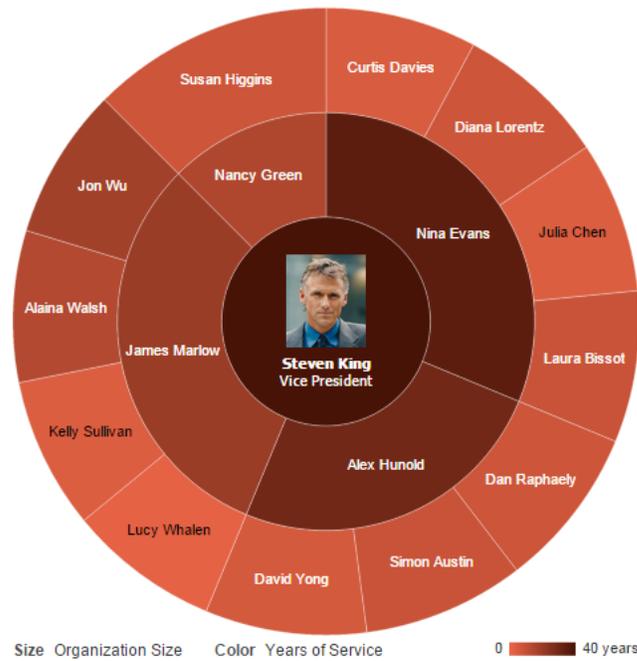


Figure 32-22 shows an example of a treemap using advanced node content.

Figure 32-22 Treemap Displaying Advanced Node Content



 **Note:**

Advanced node content is limited to `af:outputText`, `af:panelGroupLayout`, `af:spacer`, and `af:image` components. For details about configuring advanced node content, see [Configuring Treemap and Sunburst Advanced Node Content](#).

Printing and Email Support

ADF Faces allows you to output your JSF page from an ADF Faces web application in a simplified mode for printing or for emailing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scroll bars and buttons.

If a page is to be emailed, the page must be simplified so that email clients can correctly display it. For information about creating simplified pages for these outputs, see [Using Different Output Modes](#).

Active Data Support (ADS)

Treemaps and sunbursts support ADS by sending a Partial Page Refresh (PPR) request when an active data event is received. The PPR response updates the components, animating the changes as needed.

Supported ADS events include:

- Node size updates
- Node color updates
- Node label updates
- Node insertion
- Node deletion
- Enhanced node content changes

For additional information about using the Active Data Service, see [Using the Active Data Service with an Asynchronous Backend](#).

Isolation Support (Treemap Only)

Treemaps provide isolation support to focus on comparisons within groups of displayed data. Users click the **Isolate** icon that appears when the user moves the mouse over the group header to maximize the group.

[Figure 32-23](#) shows the **Isolate** icon that appears when the user moves the mouse over the South Atlantic group header.

Figure 32-23 Isolate Icon Displayed on Treemap Group Header

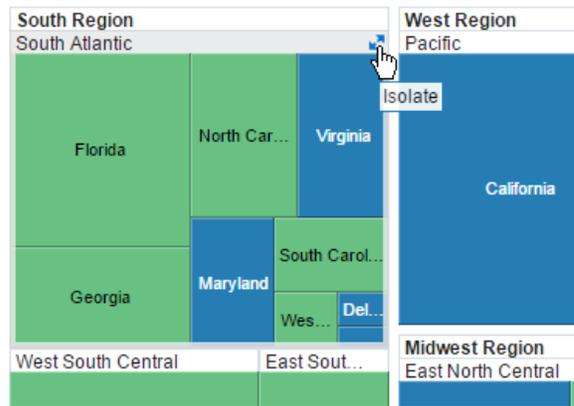
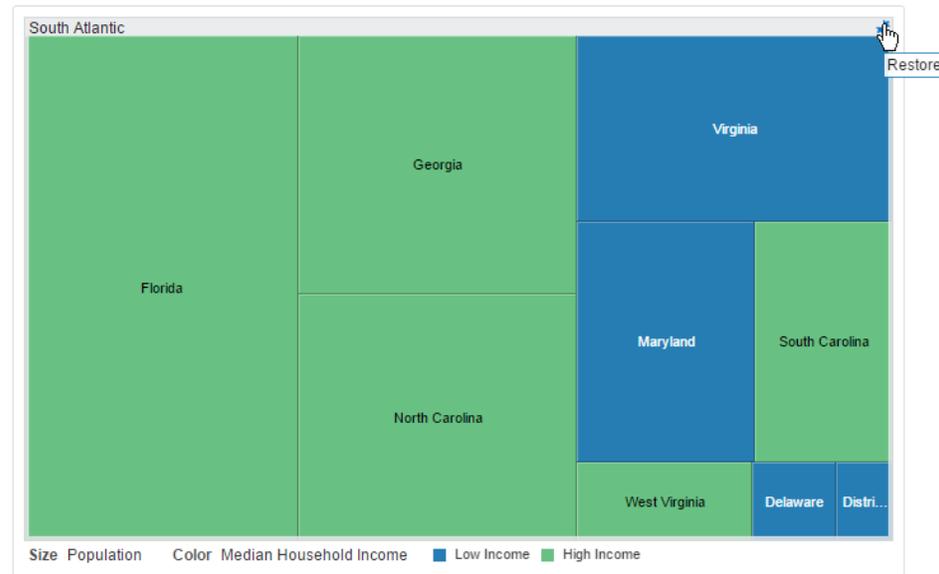


Figure 32-24 shows the treemap that is displayed when the users click the **Isolate** icon for the South Atlantic region in Figure 32-23.

To restore the treemap to the original view, users click the **Restore** icon.

Figure 32-24 Treemap Isolated on a Group

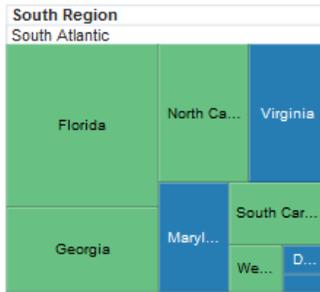


Treemap Group Node Header Customization (Treemap Only)

When the treemap displays multiple levels, the parent level is displayed in a group header. By default, the group header is displayed with a white background, and the group's title is aligned to the left in left-to-right mode and displayed with black text.

Figure 32-25 shows a portion of a treemap with node headers. In this example, the South Region and South Atlantic headers are formatted with the default formatting.

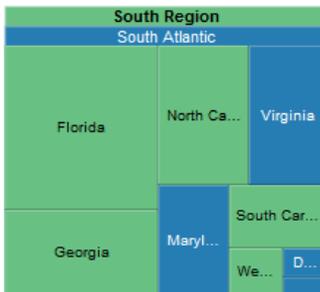
Figure 32-25 Treemap Showing Default Node Header Formatting



You can customize the headers to use the node's color, change the text alignment or customize the font.

[Figure 32-26](#) shows the same treemap with the node header formatted to use the node's color, align the title to the center and change the font size and color.

Figure 32-26 Treemap Showing Formatted Group Node Headers



In this example, the treemap nodes are displayed in red when the income levels are lower than \$50,000, and the treemap nodes are displayed in blue when the income levels are higher than \$50,000. The South Atlantic node header is displayed in blue because the color is calculated from the same rules that were used to calculate the color of the individual nodes. In this case, the income levels of all nodes contained within the South Atlantic division are higher than \$50,000. However, the South Region node header is displayed in red because it also includes the West South Central and East South Central divisions. In this case, the income levels of all nodes contained within the South Region is less than \$50,000.

Additional Functionality for Treemap and Sunburst Components

You may find it helpful to understand other ADF Faces features before you implement your `treemap` or `sunburst` component. Additionally, once you have added a treemap or sunburst to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that `treemap` and `sunburst` components can use:

- Partial page rendering: You may want a treemap or sunburst to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: When enabled, users can change the way the treemap or sunburst displays at runtime. Those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Skins and styles: You can customize the appearance of treemap or sunburst components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Accessibility: You can make your treemap and sunburst components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound treemaps and sunbursts based on how your ADF Business Components are configured. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Treemap and Sunburst Components

To use the ADF DVT Treemap and Sunburst components, add the selected component to a page using the Component Palette window. Then define the data for the treemap or sunburst and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Treemap and Sunburst Data Requirements

Treemap and sunburst components require data collections where a master-detail relationship exists between one or more detail collections and a master detail collection. Both components use the same data model as the ADF Faces `tree` component.

For more information about the `tree` component, see [Displaying Data in Trees](#).

Treemaps and sunbursts require that the following attributes be set in JDeveloper:

- `value`: the size of the node
- `fillColor`: the color of the node
- `label`: a text identifier for the node

The values for the `value` and `label` attributes must be stored in the treemap's or sunburst's data model or in classes and managed beans if you are using UI-first development. You can specify the `fillColor` values in the data model, classes, and managed beans, or declaratively in the Properties window.

[Figure 32-27](#) shows a subset of the data used to generate the treemap in [Figure 32-1](#). This is the same data used to generate the sunburst in [Figure 32-2](#).

Figure 32-27 Treemap and Sunburst Sample Data

	Value (Population)	Income (Median Household)	Color (Derived From Income)	Label
United States	301461533	51425	303030	United States
West Region	69768366	56171	3E5D1F	West Region
Pacific Division	48465072	58735	456823	Pacific
California	36308527	60392	4A6F25	California
Oregon	3727407	49033	293D14	Oregon
Hawaii	1280241	64661	56822B	Hawaii
Washington	6465755	56384	309030	Washington
Alaska	683142	64635	56822B	Alaska

In this example, United States is the `root` node with three child levels: `region`, `division`, and `state`.

In order to configure a treemap or sunburst successfully, ensure that the data adheres to the following rules:

- Each child node can have only one parent node.
- There can be no skipped levels.

To create a treemap or sunburst model in UI-first development, use classes and managed beans to define the tree node and tree model, populate the tree with data and add additional methods as needed to configure the treemap or sunburst.

See [Sample Code for Treemap and Sunburst Census Data Example](#) for sample code that defines the classes and managed beans used in both the treemap and sunburst census data examples.

Using the Treemap Component

To use the `treemap` component, add the treemap to a page and complete the additional configuration in JDeveloper.

Configuring Treemaps

The `treemap` component has configurable attributes and child components that you can add or modify to customize the display or behavior of the treemap. The prefix `dvt:` occurs at the beginning of each treemap component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure treemap child components, attributes, and supported facets in the following areas:

- **Treemap (`dvt:treemap`):** Wraps the treemap nodes. Configure the following attributes to control the treemap display.
 - **Labels:** Use the `colorLabel` and `sizeLabel` attributes to identify the color and size metrics for the treemap. Treemaps require these labels for legend display.
 - **Legend source (`legendSource`):** Use this attribute to display a legend for treemaps configured with attribute groups. Specify the id of the attribute group.
 - **Display child levels (`displayLevelsChildren`):** Specify the number of child levels to display. By default, treemaps display the root and the first two child levels.
 - **Animation:** Use the `animationOnDisplay` attribute to control the initial animation and the `animationOnDataChange` attribute to control subsequent animations. To change the animation duration from the default duration of 500 milliseconds, modify the `animationDuration` attribute.
 - **Empty text (`emptyText`):** Use the `emptyText` attribute to specify the text to display if a treemap node contains no data.
 - **Group gaps (`groupGaps`):** Specify the gaps to display between groups. By default, this attribute is set to `outer`, and the treemap displays gaps between the outer nodes only. You can remove all gaps by setting this attribute to `none` or add gaps between all groups by setting this attribute to `all`.
 - **Sorting (`sorting`):** If your treemap uses a slice and dice layout, use this attribute to sort all nodes having the same parent in descending size.
 - **Other group:** Use the `otherThreshold`, `otherThresholdBasis`, `otherColor`, and `otherPattern` attributes to aggregate child data into an **Other** node.
- **Treemap node (`dvt:treemapNode`):** child of the `treemap` component. This tag defines the size and color for each node in the treemap and is stamped for each row in the data model. If you want to vary the stamp by row, use the ADF Faces `af:switcher` component, and insert a treemap node for each row. Configure the following attributes to control the node display:
 - **value (required):** Specify the value of the treemap node. The value determines the relative size of the node within the treemap.
 - **fillColor (required):** Specify the fill color for the node in RGB hexadecimal. This value is also required for the treemap to display properly.

- `fillPattern`: Specify an optional fill pattern to use. The pattern is drawn with a white background and the foreground color uses the color specified in the `fillColor` attribute.
 - `groupLabelDisplay`: Specify where you want the group label to appear. By default this value is set to `header` which will display the label on the group header. You can also set it to `off` to turn off the label display or `node` to display the label inside the node.
 - `label`: Specify the label for the node.
 - `labelDisplay`: Specify where you want the node label to appear. By default, this attribute is set to `node` which will display the label inside the node, but you can also set it to `off` to turn off the label display.
- You can further customize the label display by setting the `labelHalign`, `labelStyle`, and `labelValign` attributes.
- **Treemap node header** (`dvt:treemapNodeHeader`): optional child of the treemap node. Add this attribute to configure the following node header attributes:
 - `isolate`: By default, this attribute is set to `on`, but you can set it to `off` to disable isolation support.
 - `labelStyle`: Specify the font style.
 - `useNodeColor`: By default, this attribute is set to `off`. Set this to `on` to display the node's color in the header.
 - `titleHalign`: Specify where you want the title to appear in the header. By default, this attribute is set to `start` which aligns the title to the left in left-to-right mode and aligns it to the right in right-to-left mode.
 - **Attribute group** (`dvt:attributeGroup`): optional child of the treemap node. Add this attribute to generate `fillColor` and `fillPattern` values automatically based on categorical bucketing or continuous classification of the data set.
 - **Supported facets**: optional children of the treemap or treemap node. The `treemap` component supports facets for displaying popup components, and the `treemap's node` component supports a content facet for providing additional detail when the treemap node's label is not sufficient.

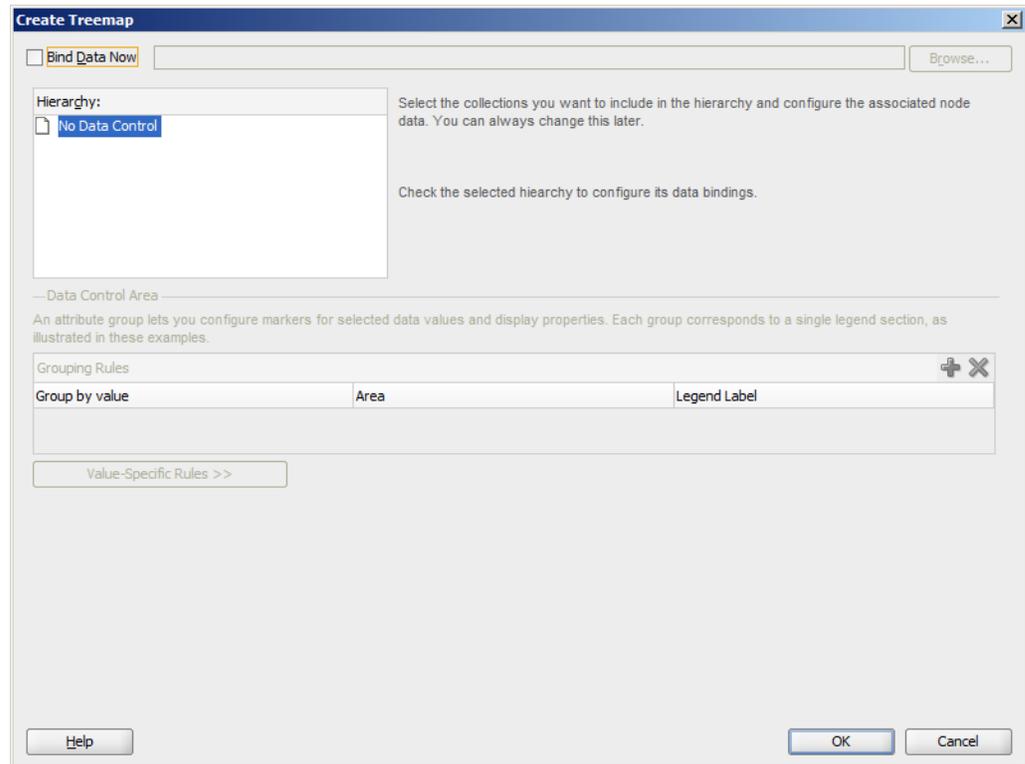
Treemaps also share much of the same functionality and tags as other DVT components. For a complete list of treemap tags, consult the *Tag Reference for Oracle ADF Faces Data Visualization Tools*. For information about additional functionality for treemap components, see [Additional Functionality for Treemap and Sunburst Components](#).

How to Add a Treemap to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a treemap to a JSF page. When you drag and drop a treemap component onto the page, a Create Treemap dialog displays.

[Figure 32-28](#) shows the Create Treemap dialog.

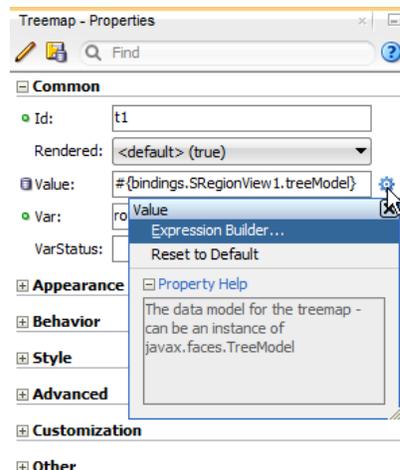
Figure 32-28 Create Treemap Dialog Using UI-First Development



If you click **OK**, the treemap is added to your page, and you can use the Properties window to specify data values and configure additional display attributes. Alternatively, you can choose to bind the data during creation and use the dialog to configure the associated node data.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. [Figure 32-29](#) shows the dropdown menu for a treemap `value` attribute.

Figure 32-29 Treemap Value Attribute Dropdown Menu



 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a treemap and the binding will be done for you. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and treemap child tags can affect functionality. For more information, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a treemap to a page:

1. In the ADF Data Visualization page of the Components window, from the Common panel, drag and drop a **Treemap** onto the page to open the Create Treemap dialog.
2. In the Create Treemap dialog, click **OK** to add the treemap to the page.
Optionally, use the dialog to bind the treemap by selecting **Bind Data Now** and navigating to the ADF data control that represents the data you wish to display on the treemap. If you choose this option, the data binding fields in the dialog will be available for editing. For help with the dialog, press F1 or click **Help**.
3. In the Properties window, view the attributes for the treemap. Use the help button to display the complete tag documentation for the `treemap` component.
4. Expand the **Appearance** section, and enter a value for the **Summary** attribute.
Enter a summary of the treemap's purpose and structure for accessibility support. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression

What Happens When You Add a Treemap to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a treemap from the Components window onto a JSF page and choose not to bind the data during creation.

The example below shows the generated code for the treemap.

```
<dvt:treemap id="t1">
  <f:facet name="multiSelectContextMenu"/>
  <f:facet name="contextMenu"/>
  <f:facet name="bodyContextMenu"/>
```

```
<dvt:treemapNode id="tn1"/>
</dvt:treemap>
```

If you choose to bind the data to a data control when creating the treemap, JDeveloper generates code based on the data model. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Using the Sunburst Component

To use the `sunburst` component, add the sunburst to a page and complete the additional configuration in JDeveloper.

Configuring Sunbursts

The `sunburst` component has configurable attributes and child components that you can add or modify to customize the display or behavior of the treemap. The prefix `dvt:` occurs at the beginning of each treemap component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure sunburst child components, attributes, and supported facets in the following areas:

- **Sunburst (`dvt:sunburst`):** Wraps the sunburst nodes. Configure the following attributes to control the sunburst display.
 - **Labels:** Use the `colorLabel` and `sizeLabel` attributes to identify the color and size metrics for the sunburst. Sunbursts require these labels for legend display.
 - **Legend source (`legendSource`):** Use this attribute to display a legend for sunbursts configured with attribute groups. Specify the id of the attribute group.
 - **Display child levels (`displayLevelsChildren`):** Specify the number of child levels to display. By default, sunbursts display the root and the first two child levels.
 - **Animation:** Use the `animationOnDisplay` attribute to control the initial animation and the `animationOnDataChange` attribute to control subsequent animations. To change the animation duration from the default duration of 500 milliseconds, modify the `animationDuration` attribute.
 - **Rotation (`rotation`):** Use this attribute to enable client-side sunburst rotation.
 - **Start angle (`startAngle`):** Specify the starting angle of the sunburst.
 - **Empty text (`emptyText`):** Use the `emptyText` attribute to specify the text to display if a sunburst node contains no data.
 - **Sorting (`sorting`):** If your treemap uses a slice and dice layout, use this attribute to sort all nodes having the same parent in descending size.
 - **Other group:** Use the `otherThreshold`, `otherThresholdBasis`, `otherColor`, and `otherPattern` attributes to aggregate child data into an **Other** node.
- **Sunburst node (`dvt:sunburstNode`):** child of the `sunburst` component. This tag defines the size and color for each node in the sunburst and is stamped for each

row in the data model. If you want to vary the stamp by row, use the ADF Faces `af:switcher` component, and insert a sunburst node for each row. Configure the following attributes to control the node display:

- `value` (required): Specify the value of the sunburst node. The value determines the relative size of the node within the sunburst.
- `fillColor` (required): Specify the fill color for the node in RGB hexadecimal. This value is also required for the sunburst to display properly.
- `fillPattern`: Specify an optional fill pattern to use. The pattern is drawn with a white background and the foreground color uses the color specified in the `fillColor` attribute.
- `label`: Specify the label for the node.
- `labelDisplay`: Specify how you want the node label to appear. By default, this attribute is set to `auto`, and the sunburst will display rotated labels inside the node if the client supports rotated text. If the client does not support rotated text, the sunburst will display horizontal labels inside the node instead. You can also set it to `off` to turn off the label display, to `on` to display horizontal labels within the nodes, or to `rotated` to display rotated labels.

 **Note:**

Rotated text is not supported on all client technologies. In particular, rotated text is not supported on clients using the Flash image format. If the client does not support rotated text and the `labelDisplay` attribute is set to `auto` or `rotated`, the sunburst will display horizontal labels within the nodes.

- `labelHalign`: Specify the label alignment for the node. By default, this attribute is set to `center` which aligns the label in the center of a node slice. You can also set this to `inner` which aligns the label to the inner side of a slice, or `outer` which aligns the label to the outer side of a slice.
- `radius`: Specify the radius of the node relative to the other nodes. By default, this attribute is set to `1`. You can specify a value or enter an EL Expression that returns the radius for the node.
- **Attribute group** (`dvt:attributeGroup`): optional child of the sunburst node. Add this attribute to generate `fillColor` and `fillPattern` values automatically based on categorical bucketing or continuous classification of the data set.
- **Supported facets**: optional children of the sunburst or sunburst node. The `sunburst` component supports facets for displaying popup components, and the `sunburst's node` component supports a content facet for providing additional detail when the sunburst node's label is not sufficient.

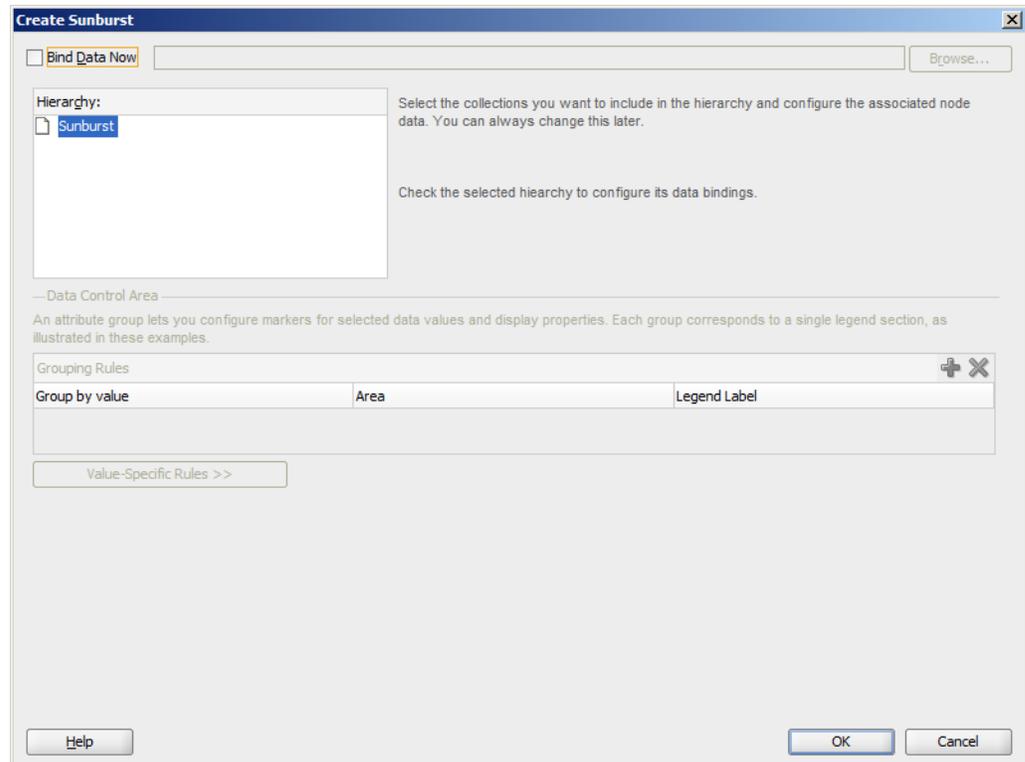
Sunbursts also share much of the same functionality and tags as other DVT components. For a complete list of sunburst tags, consult the *Tag Reference for Oracle ADF Faces Data Visualization Tools*. For information about additional functionality for sunburst components, see [Additional Functionality for Treemap and Sunburst Components](#).

How to Add a Sunburst to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a sunburst to a JSF page. When you drag and drop a sunburst component onto the page, a Create Sunburst dialog displays.

Figure 32-30 shows the Create Sunburst dialog.

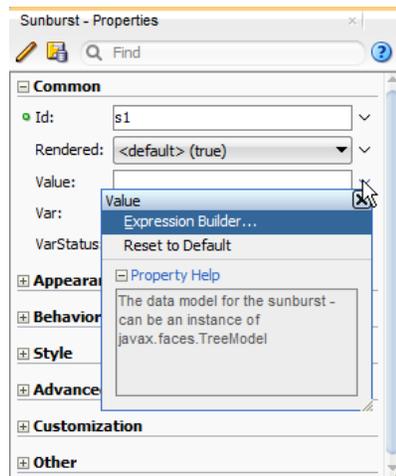
Figure 32-30 Create Sunburst Dialog Using UI-First Development



If you click **OK**, the sunburst is added to your page, and you can use the Properties window to specify data values and configure additional display attributes. Alternatively, you can choose to bind the data during creation and use the dialog to configure the associated node data.

In the Properties window you can use the dropdown menu for each attribute field to display a property description and options such as displaying an EL Expression Builder or other specialized dialogs. Figure 32-31 shows the dropdown menu for a sunburst `value` attribute.

Figure 32-31 Sunburst Value Attribute Dropdown Menu



 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a sunburst and the binding will be done for you. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how sunburst attributes and sunburst child tags can affect functionality. For more information, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a sunburst to a page:

1. In the ADF Data Visualization page of the Components window, from the Common panel, drag and drop a **Sunburst** onto the page to open the Create Sunburst dialog.
2. In the Create Sunburst dialog, click **OK** to add the sunburst to the page.

Optionally, use the dialog to bind the sunburst by selecting **Bind Data Now** and navigating to the ADF data control that represents the data you wish to display on the sunburst. If you choose this option, the data binding fields in the dialog will be available for editing. For help with the dialog, press F1 or click **Help**.

3. In the Properties window, view the attributes for the sunburst. Use the help button to display the complete tag documentation for the `sunburst` component.
4. Expand the **Appearance** section, and set a value for the **Summary** attribute.
Enter text to describe the sunburst's purpose and structure for accessibility support. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.

What Happens When You Add a Sunburst to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a sunburst from the Components window onto a JSF page and choose not to bind the data during creation.

The example below shows the generated code.

```
<dvt:sunburst id="s1">
  <f:facet name="multiSelectContextMenu"/>
  <f:facet name="contextMenu"/>
  <f:facet name="bodyContextMenu"/>
  <dvt:sunburstNode id="sn1"/>
</dvt:sunburst>
```

If you choose to bind the data to a data control when creating the sunburst, JDeveloper generates code based on the data model. For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Adding Data to Treemap and Sunburst Components

You can add data to the ADF DVT Treemap or Sunburst using UI-first development by creating classes and managed beans, and then using methods to automate the creation of the tree model and reference the data classes and beans.

How to Add Data to Treemap or Sunburst Components

Because treemaps and sunbursts use the same data model, the process for adding data to the treemap or sunburst is similar.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap or sunburst to your page. For help with adding a treemap to a page, see [How to Add a Treemap to a Page](#). For help with sunbursts, see [How to Add a Sunburst to a Page](#).

To add data to the treemap or sunburst in UI-first development:

1. Create the classes and managed beans that will define the treemap's tree model and supply the data to the treemap. See [Treemap and Sunburst Data Requirements](#) for additional information and examples. If you need help creating classes, see the "Working with Java Code" chapter of *Developing Applications with Oracle JDeveloper*. For help with managed beans, see [Creating and Using Managed Beans](#).
2. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go To Properties**.
3. In the Properties window, in the **Appearance** section, enter a value for the **DisplayLevelsChildren** attribute to change the number of child levels displayed on the treemap. By default, this value is set to 2.

For example, the treemap and sunburst in the census data example have three child levels to represent regions, divisions, and states, and you would set this value to 3 to duplicate the example.

4. In the **Common** section, set the following attributes:
 - **Value:** Specify an EL expression for the object to which you want the treemap or sunburst to be bound. This must be an instance of `org.apache.myfaces.trinidad.model.TreeModel`.

For example, reference the managed bean you created to instantiate the treemap or sunburst. In the census data example, the treemap managed bean is named `treemap`, and the census data is instantiated when the treemap is referenced. To use the census data example with a treemap, enter the following in the **Value** field for the EL expression: `#{treemap.censusData}`.

For help with creating EL expressions, see [How to Create an EL Expression](#).

- **Var:** Enter the name of a variable to be used during the rendering phase to reference each element in the treemap collection. This variable is removed or reverted back to its initial value once rendering is complete.

For example, enter `row` in the **Var** field to reference each element in the census data example.
 - **VarStatus:** Optionally, enter the name of a variable during the rendering phase to access contextual information about the state of the component, such as the collection model or loop counter information. This variable is removed or reverted back to its initial value once rendering is complete.
5. In the Structure window, right-click the **dvt:treemapNode** node or **dvt:sunburstNode** and choose **Go To Properties**.
 6. In the Common section, use the **Value** attribute's dropdown menu to choose **Expression Builder**.
 7. In the Expression Builder dialog, create the EL expression that will reference the size data for the treemap or sunburst node, using the variable you specified for the **Var** attribute when creating your component and the method you created to return the size of the node.

For example, in the census data example, the **Var** attribute is named `row` and the size is stored in the `m_size` variable which is returned by the `getSize()` method in the `TreeNode` class shown in the first code sample in [Sample Code for Treemap and Sunburst Census Data Example](#) in Appendix F. To reference the size data in the census data example, create the following expression: `#{row.size}`.

8. In the Properties window, expand the **Appearance** section and enter values for the following attributes:

- **FillColor:** Specify the fill color of the node. You can enter the color in RGB hexadecimal or use the attribute's dropdown menu to choose **Expression Builder** and create an EL expression.

For example, you could enter #FF0000 to set the node's fill color to red. However, you might want your treemap or sunburst node to change color based on the color metric. In the census data example in [Figure 32-1](#), the fill color is calculated from income data.

The example below shows the sample method used by the census data example. To reference this example in the Expression Builder, create the following expression: `#{row.color}`.

```
import java.awt.Color;

private static Color getColor(double value, double min, double max)
{
    double percent = Math.max((value - min) / max, 0);
    if(percent > 0.5) {
        double modifier = (percent - 0.5) * 2;
        return new Color((int)(modifier*102), (int)(modifier*153), (int)
(modifier*51));
    }
    else {
        double modifier = percent *2;
        return new Color((int)(modifier*204), (int)(modifier*51), 0);
    }
}
```

- **Label:** Specify the node's label. You can enter text or use the attribute's dropdown menu to choose **Expression Builder** and create an EL expression.

For example, the census data example uses a method that converts the node data into strings for label display. See the last code sample in [Sample Code for Treemap and Sunburst Census Data Example](#) in Appendix F for the `convertToString()` method. The `TreeNode` class uses the output from the `convertToString()` method to set the `text` variable which is used for the label display. To reference this example in the Expression Builder dialog, create the following expression: `#{row.text}`.

 **Note:**

You can also use attribute groups to set the `fillColor` and `label` attribute. Attribute groups are optional, but you must use them if you want your treemap or sunburst to change color or pattern based on a given condition, such as high versus low income. For information about configuring attribute groups, see [How to Configure Treemap and Sunburst Discrete Attribute Groups](#).

What You May Need to Know about Adding Data to Treemaps and Sunbursts

The examples in this chapter use classes and managed beans to provide the data to the treemap and sunburst. If your application uses the Fusion technology stack, then you can use data controls to create a sunburst and the binding will be done for you.

For more information, see the "Creating Databound Hierarchy Viewer, Treemap, and Sunburst Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Alternatively, if you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Customizing Treemap and Sunburst Display Elements

ADF DVT Treemaps and Sunbursts are highly customizable and allow many configurations to suit business data. You can configure elements such as attribute groups, legends, labels, patterns, skins, sizes, and even animation. You can also aggregate smaller data contributors and change the order of nodes.

Configuring Treemap and Sunburst Display Size and Style

You can configure the treemap or sunburst's size and style using the `inlineStyle` or `styleClass` attributes. Both attributes are available in the **Style** section in the Properties window for the `dvt:treemap` or `dvt:sunburst` component.

Using these attributes, you can customize stylistic features such as fonts, borders, and background elements.

For additional information about using the `inlineStyle` or `styleClass` attributes, see [Customizing the Appearance Using Styles and Skins](#).

The page containing the treemap or sunburst may also impose limitations on the ability to change the size or style. For more information about page layouts, see [Organizing Content on Web Pages](#).

What You May Need to Know About Skinning and Configuring Treemap and Sunburst Display Size and Style

Treemaps and sunbursts also support skinning to customize the color and font styles for the top level components as well as the nodes, node headers, and icons used for treemap isolation and sunburst expansion and collapse. You can also use skinning to define the styles for a treemap or sunburst node or a treemap node header when the user hovers the mouse over or selects a node or node header. If the node or node

header is drillable, you can use skinning to define the styles when the user hovers the mouse over or selects it.

The example below shows the skinning key for a sunburst configured to show the node's text in bold when the user selects it.

```
af|dvt-sunburstNode::selected
{
  -tr-font-weight: bold;
}
```

For the complete list of treemap and sunburst skinning keys, see the *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces Skin Selectors*. For additional information about customizing your application using skinning and styles, see [Customizing the Appearance Using Styles and Skins](#).

Configuring Pattern Display

You can configure the treemap or sunburst node to display patterns using the `fillPattern` attribute.

The available patterns are:

- none (default)
- smallChecker
- smallCrosshatch
- smallDiagonalLeft
- smallDiagonalRight
- smallDiamond
- smallTriangle
- largeChecker
- largeCrosshatch
- largeDiagonalLeft
- largeDiagonalRight
- largeDiamond
- largeTriangle

To configure the treemap or sunburst node to display patterns, specify the `fillPattern` attribute on the **dvt:treemapNode** or **dvt:sunburstNode** node. You can also use discrete attribute groups to specify the fill pattern. For more information about discrete attribute groups, see [How to Configure Treemap and Sunburst Discrete Attribute Groups](#).

Configuring Treemap and Sunburst Attribute Groups

Use attribute groups to generate stylistic attribute values such as colors or shapes based on categorical bucketing of a data set. Treemaps and sunbursts support both discrete and continuous attribute groups for setting the color and pattern of the child nodes.

Use a discrete attribute group if you want the color or pattern to depend upon a given condition, such as high or low income levels. Use the continuous attribute group if you want the color to change gradually between low and high values.

How to Configure Treemap and Sunburst Discrete Attribute Groups

Configure discrete attribute groups by adding the `dvt:attributeGroups` tag to your treemap or sunburst and defining the conditions under which the color or pattern will be displayed.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).

To configure a treemap or sunburst discrete attribute group:

1. In the Structure window, right-click the `dvt:treemapNode` or `dvt:sunburstNode` node and choose **Insert Inside Component Node > Attribute Groups**.

For example, to configure a treemap discrete attribute group, right-click the `dvt:treemapNode` node and choose **Insert Inside Treemap Node > Attribute Groups**.

2. Right-click the `dvt:attributeGroups` element and choose **Go to Properties**.
3. In the Properties window, expand the **Appearance** section.
4. From the **Value** attribute's dropdown menu, choose **Expression Builder** and create an expression that references the color metric and the condition that will control the color display.

For example, if you want your treemap to display different colors for median income levels higher or lower than \$50,000 as shown in [Figure 32-6](#), create an expression similar to the following expression for the **Value** field:

```
#{row.income > 50000}
```

For help with creating EL expressions, see [How to Create an EL Expression](#).

5. From the **Label** attribute's dropdown menu, choose **Expression Builder** and create an expression for the legend that describes what the discrete colors or patterns represent.

For example, to let the user know that the colors represent high and low median income levels, create an expression similar to the following expression for the **Label** field:

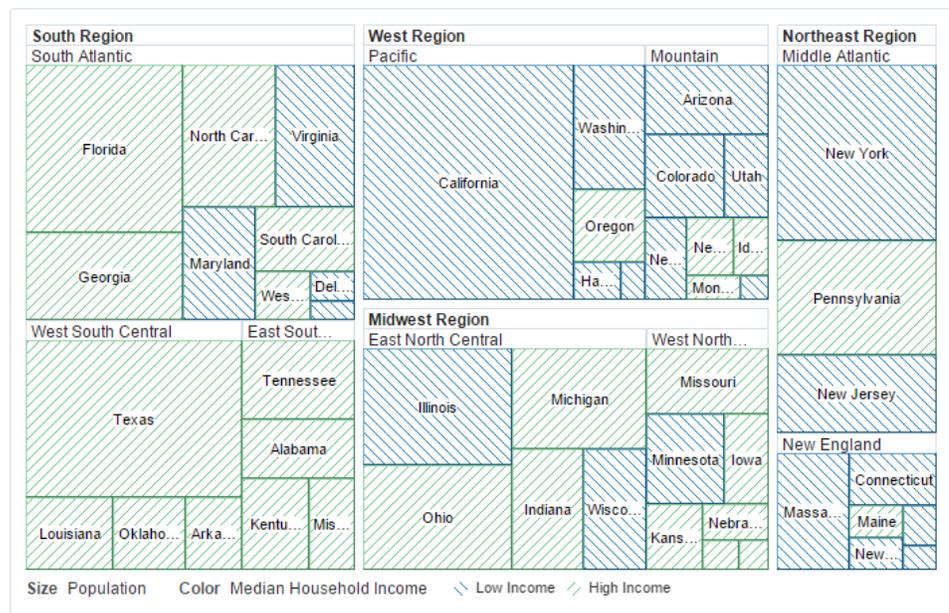
```
#{row.income > 50000 ? 'High Income' : 'Low Income'}
```

6. From the **Type** attribute's dropdown menu, choose **Edit**.
7. From the Edit Property dialog, choose color, pattern, or both, and click **OK**.

If you choose both color and pattern and build the page now, the treemap or sunburst will use default colors and patterns for the discrete attribute group.

Figure 32-32 shows the treemap that displays if you accept the default colors and patterns in the census data example.

Figure 32-32 Treemap Discrete Attribute Group with Default Colors and Patterns



8. Optionally, to change the attribute group's default colors, do the following:
 - a. In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups > Attribute Match Rule**.
The **dvt:attributeMatchRule** tag is used to replace an attribute when the data matches a given condition. In the census data example, the condition is median income higher than \$50,000.
 - b. Right-click the **dvt:attributeMatchRule** element and choose **Go to Properties**.
 - c. In the **Group** field, enter **true** if you want the color to display when the condition is true, or enter **false** if you want the color to display when the condition is false.
For example, enter **true** to choose the color to display in the census data example when the median income level is higher than 50000.
 - d. In the Structure window, right-click the **dvt:attributeMatchRule** element and choose **Insert Inside Match Rule > Attribute**.

- e. In the Insert Attribute dialog, enter `color` for the **name** field and a color in the **value** field, and click **OK**.
The value field accepts a six-digit RGB hexadecimal value. For example, to set the value to green, enter the following in the **value** field: `#00AA00`.
 - f. Repeat step 8.a through step 8.e if you want to change the default color for the other half of the condition.
For example, add another match rule to define the color that displays when the income is under 50000, and set the **Group** field to `false`.
9. Optionally, to change the attribute group's default patterns, do the following:
 - a. In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups > Attribute Match Rule**.
 - b. Right-click the **dvt:attributeMatchRule** element and choose **Go to Properties**.
 - c. In the **Group** field, enter `true` if you want the pattern to display when the condition is true, or enter `false` if you want the pattern to display when the condition is false.
 - d. In the Structure window, right-click the **dvt:attributeMatchRule** element and choose **Insert Inside Attribute Match Rule > Attribute**.
 - e. In the Insert Attribute dialog, enter `pattern` for the **Name** field and a supported pattern in the **Value** field, and click **OK**.
For example, enter `smallDiamond` in the **Value** field to change the pattern to small diamonds. For the list of available patterns, see [Configuring Pattern Display](#).
 - f. Repeat step 9.a through step 9.e if you want to change the default color for the other half of the condition.
For example, add another match rule to define the color that displays when the income is under 50000, and set the **Group** field to `false`.

The example below shows the code on the JSF page if you configure a discrete attribute group for the treemap shown in [Figure 32-6](#).

```
<dvt:treemap id="t1" summary="SampleTreemap" value="#{treemap.censusData}"
  var="row" colorLabel="Median Household Income" sizeLabel="Population"
  displayLevelsChildren="3" emptyText="No Data to Display"
  legendSource="ag1">
  <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}">
    <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
      label="#{row.income > 50000 ? 'High Income' : 'Low
Income'}"
      type="color">
      <dvt:attributeMatchRule id="amr1" group="true">
        <f:attribute name="color" value="#00AA00"/>
      </dvt:attributeMatchRule>
      <dvt:attributeMatchRule id="amr2" group="false">
        <f:attribute name="color" value="#AA0000"/>
      </dvt:attributeMatchRule>
    </dvt:attributeGroups>
    <f:facet name="content"/>
  </dvt:treemapNode>
</dvt:treemap>
```

How to Configure Treemap or Sunburst Continuous Attribute Groups

Configure continuous attribute groups by adding the `dvt:attributeGroups` tag to your treemap or sunburst and defining the colors to be displayed at the minimum and maximum levels of the data range. The attribute group will use the data to determine the data range and display labels in the legend with corresponding values, but you can also configure the attribute group to use different ranges or labels.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).

To configure a treemap or sunburst continuous attribute group:

1. In the Structure window, right-click the `dvt:treemapNode` or `dvt:sunburstNode` node and choose **Insert Inside Component Node > Attribute Groups**.

For example, to configure a treemap continuous attribute group, right-click the `dvt:treemapNode` node and choose **Insert Inside Treemap Node > Attribute Groups**.

2. Right-click the `dvt:attributeGroups` element and choose **Go to Properties**.
3. In the Properties window, expand the **Appearance** section.
4. From the **Value** attribute's dropdown menu, choose **Expression Builder** and enter an expression that references the color metric.

For example, to specify an EL expression that returns the income data from the census example, choose **Expression Builder** and enter the following value in the **Value** field: `#{row.income}`. For help with creating EL expressions, see [How to Create an EL Expression](#).

5. In the **Type** field, enter `color`.
6. In the **AttributeType** field, use the attribute's dropdown menu to choose `continuous`.
7. Optionally, set values for the following minimum or maximum range and labels:
 - **MinValue**: Enter the minimum boundary for the range. Alternatively, choose **Expression Builder** from the attribute's dropdown menu and enter the expression that returns the minimum boundary.

For example, enter 35000 in the **MinValue** field to set the lower boundary of the range to 35,000.

- **MaxValue:** Enter the maximum boundary for the range. Alternatively, choose **Expression Builder** from the attribute's dropdown menu and enter the expression that returns the maximum bound.
- **MinLabel:** Enter the label for the minimum value to be displayed in the legend. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.

For example, enter \$35000 in the **MinLabel** field to set the label displayed in the legend to \$35000.

- **MaxLabel:** Enter the label for the maximum value to be displayed in the legend. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
8. To define the colors used for the minimum and maximum bounds of the range, do the following:

- In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups > Attribute**.
- In the Insert Attribute dialog, enter `color1` for the name and a value for the minimum boundary, and click **OK**.

The value field accepts a six-digit RGB hexadecimal value. For example, to set the value of the minimum bound to a dark green, which is the color used in the attribute group in [Figure 32-1](#), enter the following in the **value** field:
#14301C.

- In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups > Attribute**.
- In the Insert Attribute dialog, enter `color2` for the name and a value for the maximum boundary, and click **OK**.

The value field accepts a six-digit RGB hexadecimal value. For example, to set the value of the maximum bound to a light green, which is the color used in the attribute group in [Figure 32-1](#), enter the following in the **value** field:
#68C182.

[Figure 32-9](#) shows the code on the JSF page if you configure the continuous attribute group shown in [Figure 32-1](#).

```
<dvt:treemap id="t1" summary="SampleTreemap" value="{treemap.censusData}"
    var="row" colorLabel="Median Household Income"
    sizeLabel="Population"
    displayLevelsChildren="3" emptyText="No Data to Display"
    legendSource="ag1">
  <dvt:treemapNode id="tn1" value="{row.size}" label="{row.text}">
    <dvt:attributeGroups id="ag1" value="{row.income}" type="color"
      attributeType="continuous" minValue="35000"
      maxValue="70000" minLabel="$35000"
      maxLabel="$70000">
      <f:attribute name="color1" value="#14301C"/>
      <f:attribute name="color2" value="#68C182"/>
    </dvt:attributeGroups>
  </dvt:treemapNode>
</dvt:treemap>
<f:facet name="content"/>
```

```
</dvt:treemapNode>  
</dvt:treemap>
```

What You May Need to Know About Configuring Attribute Groups

If you use the **Other** node to aggregate nodes for display, the **Other** node will not use the color or pattern of the configured attribute group.

For more information, see [What You May Need to Know About Configuring the Treemap and Sunburst Other Node](#).

How to Configure Treemap and Sunburst Legends

Legends display automatically when you specify values for the following attributes:

- `sizeLabel`: Specify the text that describes the size metric of the component. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
- `colorLabel`: Specify the text that describes the color metric of the component. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
- `legendSource`: Optionally, specify the id of the attribute group used in the treemap or sunburst display.

If your treemap or sunburst does not use attribute groups, the legend display will be limited to the text descriptions that you specified for the size and color labels.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).

To configure a treemap or sunburst legend:

1. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section.
3. In the **SizeLabel** field, enter the text that the legend will display to describe the size metric.

For example, enter `Population` in the **SizeLabel** field to indicate that the size of the nodes in the treemap or sunburst is based on population.

You can also use the dropdown menu to choose a text resource or EL expression from the Expression Builder dialog. For example, to specify an EL expression that returns the size from the census data example, choose **Expression Builder** and enter the following value in the **SizeLabel** field: `#{row.size}`. For help with creating EL expressions, see [How to Create an EL Expression](#).

4. In the **ColorLabel** field, enter the text that the legend will display to describe the color metric.

For example, enter `Median Household Income` in the **ColorLabel** field to indicate that the size of the nodes in the treemap or sunburst is based on population.

Alternatively, use the dropdown menu to enter a text resource or select an expression from the Expression Builder. For example, to specify an EL expression that returns the color from the census data example, choose **Expression Builder** and enter the following value in the **ColorLabel** field: `#{color.size}`.

5. If your treemap or sunburst uses attribute groups, reference the id of the `attributeGroups` component as follows:
 - a. From the **LegendSource** property's dropdown menu, choose **Edit**.
 - b. In the Edit Property: LegendSource dialog, expand each component and locate the `attributeGroups` component.
 - c. Select the `attributeGroups` component and click **OK**.

Configuring the Treemap and Sunburst Other Node

Use the **Other** node to aggregate smaller data sets visually into one larger set for easier comparison. You can aggregate the data sets based on the size of the node's parent or the size of the treemap or sunburst component's root node.

How to Configure the Treemap and Sunburst Other Node

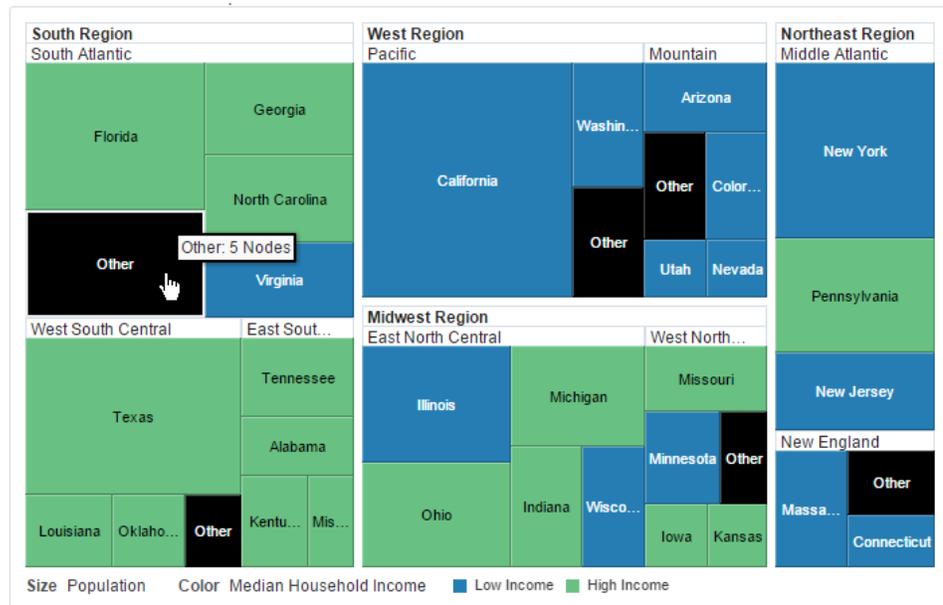
Configure the treemap or sunburst **Other** node by setting values for the following attributes:

- `otherThreshold`: Specify the percentage of the parent or root node under which a node would be aggregated into the **Other** node. Valid values range from 0 (default) to 1.

For example, a value of `0.1` would cause all nodes that are less than 10% of their parent's value to be aggregated into the **Other** node. In [Figure 32-17](#), the `otherThreshold` is set to `.08` or eight percent which aggregated the **South Carolina, Delaware, West Virginia, and District of Columbia** nodes in the South Atlantic region.

If you increase the value to `0.1` or 10%, the **Maryland** node is added to the aggregation. [Figure 32-33](#) shows the same treemap with the `otherThreshold` attribute set to `0.1`.

Figure 32-33 Treemap Showing Other Node With otherThreshold Set to 10 Percent



- `otherThresholdBasis`: Specify the basis used to apply the `otherThreshold` value as either a percentage of the root node or as a percentage of the node's parent node. Valid values are:
 - `parentPercentage`: Nodes are compared against the size of the parent node when applying `otherThreshold`. This is the default behavior and the setting used for the figures in this section.

For example, if one of the nodes in the treemap has a parent in the **Other** node, the child node will also be aggregated into the **Other** node when you drill down to its level, regardless of the node's value. If two child nodes have the same value but one node's parent is in the **Other** node, that child node will be aggregated into the **Other** node.
 - `rootPercentage`: Nodes are compared against the size of the treemap or sunburst's root node when applying `otherThreshold`.

For example, if you set `otherThreshold` to 0.1, all nodes whose values are less than 10% of the total sunburst display will be included in **Other** regardless of the size of their parents.
- `otherColor`: Specify a reference to a method that takes the `RowKeySet` of all nodes contained within the current **Other** node and returns a `String` for the color of the **Other** node.

For example, the census data example uses a method to calculate the mean income of all the nodes contained within the **Other** node. If the mean household income is less than 50,000, the method returns the same color value used to display low income as the non-aggregated nodes in the treemap. Notice how the color changed on the **Other** node in [Figure 32-33](#) to reflect the higher mean income when the **Maryland** node is included in the **Other** node.

The example below shows the sample method to specify the `otherColor` value based on the mean income in the census data example.

```
import org.apache.myfaces.trinidad.model.RowKeySet;
import org.apache.myfaces.trinidad.model.TreeModel;

public String otherColor(RowKeySet set) {
    // The color should be the mean income of the contained regions.
    Note that it should actually
    // be the median, but we can't calculate that with the available
    information.
    TreeModel tree = getCensusRootData();
    // Loop through and get the population + average income
    double population = 0;
    double average = 0;
    for(Object rowKey : set) {
        CensusData.CensusTreeNode item = (CensusData.CensusTreeNode)
tree.getRowData(rowKey);
        population += item.getSize().doubleValue();
        average += item.getSize().doubleValue() * item.getIncome();
    }
    // Calculate the average
    average = average / population;
    // Match the attr groups used by the demos
    return average > 50000 ? "#CC3300" : "#003366";
}
```

- `otherPattern`: Optionally, specify a reference to a method that takes the `RowKeySet` of all nodes contained within the current **Other** node and returns a `String` for the pattern of the **Other** node.

The example below shows the sample code for a method that sets the pattern fill to `smallDiamond` on the **Other** node.

```
import org.apache.myfaces.trinidad.model.RowKeySet;
public String otherPattern(RowKeySet rowKeySet) {
    return "smallDiamond";
}
```

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information

about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).

- Create the method that takes the `RowKeySet` of all nodes contained within the current **Other** node and returns a `String` for the color of the **Other** node.

To use the United States census data example, add the sample `otherColor()` method in the example above to a managed bean.

If you need help with managed beans, see [Creating and Using Managed Beans](#).

- Optionally, create the method that takes the `RowKeySet` of all nodes contained within the current **Other** node and returns a `String` for the pattern of the **Other** node.

To use the United States census data example, add the sample `otherPattern()` method in the example above to a managed bean.

To add the Other node to a treemap or sunburst:

1. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
2. In the Properties window, expand the **Other** section and enter a value for the following attributes:
 - **OtherThreshold**: Enter the percentage of nodes to be aggregated as a value between 0 and 1.
 - **OtherThresholdBasis**: To change the default basis used to apply the `otherThreshold` value to a percentage of the size of the sunburst or treemap component's root node, choose `rootPercentage` from the dropdown list.
 - **OtherColor**: Choose **Edit** from the dropdown menu and select the managed bean and method that sets the `otherColor` attribute.

For example, for a managed bean named `treemap` and a method named `otherColor`, enter the following in the **OtherColor** field:
`#{treemap.otherColor}`.

- **OtherPattern**: Choose **Edit** from the dropdown and select the managed bean and method that sets the `otherPattern` attribute.

For example, for a managed bean named `treemap` and a method named `otherPattern`, enter the following in the **OtherPattern** field:
`#{treemap.otherPattern}`.

What You May Need to Know About Configuring the Treemap and Sunburst Other Node

Because the **Other** node is an aggregation of individual nodes, its behavior will be different than other treemap and sunburst child nodes when managing children, attribute groups, selection, tooltips, and popup support.

Specifically, you should be aware of the following differences:

- Child nodes: Children of the aggregated nodes are not displayed unless you set the treemap or sunburst component's `otherThresholdBasis` attribute.
- Other node display with attribute groups: If you use attribute groups to specify a color or pattern, that color or pattern will not be displayed on the **Other** node. If you want the Other node to display the same color or pattern as the attribute

group, you must create methods in a managed bean to return a color or pattern that makes sense.

- Selection behavior: **Other** nodes are not selectable if you change node selection support from the default value of multiple selection to single node selection.
- Tooltips: Tooltips display the number of nodes within the **Other** node and are not customizable.
- Popups: By default, popups will not display on the **Other** node.

When a user invokes a popup on a node, that node is made current on the component (and its model), allowing the application to determine context. Treemaps and sunbursts use the `af:showPopupBehavior` tag to determine context, but this tag does not support making multiple nodes current. If you want your treemap or sunburst to display a popup on the **Other** node, you must create a method in a managed bean that calls the `getPopupContext()` method on the `UITreemap` or `UISunburst` component to determine the context of the aggregated nodes.

Configuring Treemap and Sunburst Sorting

Sorting is enabled by default if your treemap or sunburst uses the **Other** node. Otherwise you must enable it by setting the `dvt:treemap` or `dvt:sunburst` `sorting` attribute to `on` in the Properties window.

Treemaps support sorting in the slice and dice layouts only.

Configuring Treemap and Sunburst Advanced Node Content

You can configure advanced node content by defining a content facet on the treemap or sunburst node.

Both treemaps and sunbursts support the following Oracle Application Development Framework tags:

- `af:image`
- `af:outputText`
- `af:panelGroupLayout`
- `af:spacer`

Only a single child is supported for layout reasons, and you must use `af:panelGroupLayout` to wrap multiple child components. Interactive behaviors are also not supported for components within this facet.

How to Add Advanced Node Content to a Treemap

You can configure advanced node content on a treemap by defining the `content` facet on the `dvt:treemapNode` node.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap to your page. For more information, see [How to Add a Treemap to a Page](#).

To add advanced node content to a treemap:

1. In the Structure window, expand the **dvt:treemapNode**.
2. To configure the facet, in the Structure window, right-click the **f:facet - content** node and choose to **Insert Inside f:facet - content** one of the following:

- **Image**
- **Output Text**
- **Panel Group Layout**
- **Spacer**

To insert more than one component, choose the **Panel Group Layout** and add the image, output text, or spacers as required by your application. For help with configuring panel group layouts, see [How to Use the panelGroupLayout Component](#).

For help with configuring images and output text, see [Using Output Components](#).

How to Add Advanced Root Node Content to a Sunburst:

Configure advanced node content on a sunburst by defining the `rootContent` facet on the **dvt:sunburstNode** node.

Before you begin:

It may be helpful to have an understanding of sunburst attributes and child tags can affect functionality. For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a sunburst to your page. For more information, see [How to Add a Sunburst to a Page](#).

To add advanced root node content to a sunburst:

1. In the Structure window, expand the **dvt:sunburst** node.
2. To configure the facet, in the Structure window, right-click the **f:facet - content** node and choose to **Insert Inside f:facet - rootContent** one of the following:

- **Image**
- **Output Text**
- **Panel Group Layout**
- **Spacer**

To insert more than one component, choose the **Panel Group Layout** and add the image, output text, or spacers as required by your application. For help with configuring panel group layouts, see [How to Use the `panelGroupLayout` Component](#).

For help with configuring images and output text, see [Using Output Components](#).

What You May Need to Know About Configuring Advanced Node Content on Treemaps

Treemaps are meant to display two dimensions of data using size and color. Node content should be used to identify the treemap node, such as with labels or images, and should not be relied upon to display many additional dimensions of data. Applications should consider using popups for additional content since they will not have aspect ratio or small size issues like treemap nodes.

How to Configure Animation in Treemaps and Sunbursts

Treemaps and sunbursts support multiple types of animations. By default, no animation is displayed, but you can add animation to the treemap or sunburst when it initially displays. You can also customize the animation effects when a data change occurs on the component.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).

To add animation effects to a treemap or sunburst:

1. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and set a value for the following attributes:
 - **AnimationDuration**: Specify the duration of the animation in milliseconds. The default value is 500. For data changes, the animation occurs in stages, and the default value is 500 for each stage of the animation.
 - **AnimationDisplay**: Use the dropdown menu to specify the type of animation to apply when the component is initially rendered. By default, this is set to `none`.
 - **AnimationOnDataChange**: Use the dropdown menu to specify the type of animation to apply when data is changed in the treemap or sunburst. By default, this is set to `activeData` for Active Data Service data change events.

For treemap and sunburst, the `auto` type is recommended because it will apply animation for both Partial Page Refresh and Active Data Service Events.

Table 32-1 shows the list of supported animation effects.

Table 32-1 Treemap and Sunburst Animation Effects

Animation Effect	AnimationOnDisplay	AnimationOnDataChange
<code>none</code>	x	x
<code>activeData</code>		x
<code>alphaFade</code>	x	x
<code>auto</code>		x
<code>cubeToLeft</code>		x (treemap only)
<code>cubeToRight</code>		x (treemap only)
<code>fan</code>	x (sunburst only)	
<code>flipLeft</code>		x (sunburst only)
<code>flipRight</code>		x (sunburst only)
<code>slideToLeft</code>		x
<code>slideToRight</code>		x
<code>transitionToLeft</code>		x
<code>transitionToRight</code>		x
<code>zoom</code>	x	x

Configuring Labels in Treemaps and Sunbursts

Treemaps and sunbursts support customization of label display for the following elements:

- `colorLabel` and `sizeLabel`: These labels are used in the legend display. For additional information about configuring these labels, see [How to Configure Treemap and Sunburst Legends](#).
- `treemapNodeHeader`: The title displayed in treemap node headers is configurable. For additional information about customizing the treemap node header title, see [Configuring Treemap Node Headers and Group Gap Display](#).
- node labels: You can configure the size, style, and display of node labels on both treemaps and sunbursts. The options for configuration are slightly different between the components, due to the differences in layouts.

How to Configure Treemap Leaf Node Labels

You can configure treemap node labels by setting label attributes on the treemap node.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap to your page. For more information, see [How to Add a Treemap to a Page](#).

To configure treemap leaf node labels:

1. In the Structure window, right-click the **dvt:treemapNode** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and set a value for the following attributes:
 - **LabelDisplay**: Use the dropdown menu to specify whether or not labels are displayed on the leaf nodes. The default is `node` which displays the label inside the leaf node. To turn off the label display, choose `off`.
 - **LabelHalign**: Use the dropdown menu to specify the horizontal alignment for labels displayed within the node. The default value is `center`. To align the title to the left in left-to-right mode and to the right in right-to-left mode, set this value to `start`. You can also set this to `end` which aligns the title to the right in left-to-right mode and to the left in right-to-left mode.
 - **LabelValign**: Use the dropdown menu to specify the vertical alignment for labels displayed within the node. The default value is `center`. You can change this to `top` or `bottom`.
 - **LabelStyle**: Specify the font style for the label displayed in the header. This attribute accepts CSS style attributes such as `font-size` or `color`.

For example, to change the size of the title to 14 pixels and the color to white, enter the following value for **LabelStyle**:

```
font-size:14px;color: #FFFFFF
```

For the complete list of CSS attributes, visit the World Wide Web Consortium's web site at:

<http://www.w3.org/TR/CSS21/>

- **GroupLabelDisplay**: Use the dropdown menu to specify the label display behavior for group nodes. The default value is `header` which will display the group node label in the node header. You can also set this to `off` or to `node` which will display the label inside the node.

How to Configure Sunburst Node Labels

You can configure sunburst node labels by setting and customizing label attributes on the sunburst node.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a sunburst to your page. For more information, see [How to Add a Sunburst to a Page](#).

To configure sunburst node labels:

1. In the Structure window, right-click the **dvt:sunburstNode** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and set a value for the following attributes:

- **LabelStyle**: Specify the font style for the label displayed in the header. This attribute accepts CSS style attributes such as `font-size` or `color`.

For example, to change the size of the title to 14 pixels and the color to white, enter the following value for **LabelStyle**:

```
font-size:14px;color: #FFFFFF
```

For the complete list of CSS attributes, visit the World Wide Web Consortium's web site at:

<http://www.w3.org/TR/CSS21/>

- **LabelDisplay**: Use the dropdown menu to specify the label display for the nodes. The default value is `auto` which displays rotated labels within the nodes if the client's environment supports rotated text. If the client's environment does not support rotated text, the sunburst will display horizontal text inside the node. You can also set this to `off` to turn off the label display, to `on` to display horizontal labels within the nodes, or to `rotated` to display rotated labels within the nodes.

 **Note:**

If the `labelDisplay` attribute is set to `rotated` and the client's environment does not support rotated text, the sunburst will display horizontal labels within the nodes. This is the same behavior as the default `auto` setting.

- **LabelHalign**: Use the dropdown menu to specify the alignment of node labels. The default value is `center` which aligns the labels in the center of the node slices. You can also set this to `inner` which aligns the label to the inner side of a slice or `outer` which aligns the label to the outer side of a slice.

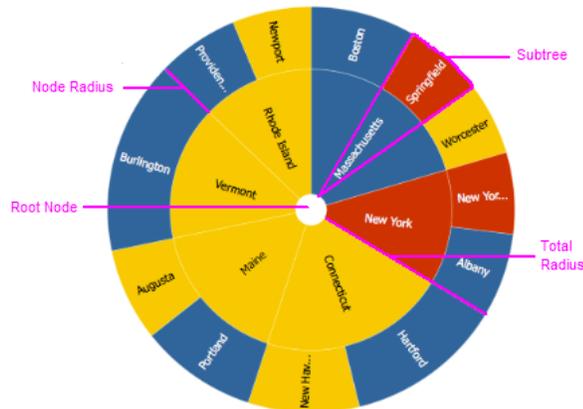
Configuring Sunburst Node Radius

By default, sunbursts allocate the same amount of display area for each node in the sunburst. If you want to increase the display area allocated to one of the nodes, set the node's `radius` attribute.

[Figure 32-34](#) shows a sunburst configured with two nodes, where the inner node represents states and the outer node represents cities within those states. The node

radius is the straight line separating each member of the node and is a segment of the sunburst's total radius.

Figure 32-34 Sunburst Showing a Node With Radius Set to 2



A sunburst's total radius equals the radius of the longest subtree in the sunburst. A subtree represents the path from the root node to the leaf node and may consist of one or more nodes with one or more layers. In [Figure 32-34](#), the subtree representing Massachusetts + Springfield is highlighted.

A sunburst's node radius is set to 1 by default, and a sunburst with two nodes representing two layers has a total radius of 2. To increase the display area for one of the nodes, increase the value of its radius. In [Figure 32-34](#), the inner node is configured with its radius set to 2, and the leaf node's radius is set to 1. In this example, the sunburst's total radius is 3, and the inner node will use 2/3 of the available display area.

Note:

If your sunburst defines a single node for all layers in the sunburst, setting the radius attribute will have the effect of increasing the size of the root node's display to match the size of the other layers. If you want to vary the layers, use the ADF Faces `af:switcher` component, and insert a `f:facet` containing a `dvt:node` for each row.

How to Configure a Sunburst Node Radius

You can use the Properties window to set a value for the sunburst node radius.

Before you begin:

It may be helpful to have an understanding of how sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes. For more information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a sunburst to your page. For more information, see [How to Add a Sunburst to a Page](#).

To configure a sunburst node radius:

1. In the Structure window, right-click the **dvt:sunburstNode** node and choose **Go to Properties**.
2. In the Properties window, expand the **Other** section.
3. In the **Radius** field, enter a numeric value for the radius or use the attribute's dropdown menu to enter an EL expression that returns the radius value.

What You May Need to Know about Configuring the Sunburst Node Radius

When you change the sunburst's node radius, node labels may no longer display properly. You can adjust the alignment of the node labels using the `labelHalign` attribute.

For additional information, see [How to Configure Sunburst Node Labels](#).

Configuring Treemap Node Headers and Group Gap Display

Treemap node headers are displayed by default whenever there are two or more child levels in the treemap. Configure the node header if you wish to change the default display.

Group gaps are displayed between the outer group nodes by default. Configure group gaps if you wish to change the way group gaps are displayed between the nodes.

How to Configure Treemap Node Headers

Configure treemap node headers by adding the `dvt:treemapNodeHeader` element to your treemap node and setting values for the following attributes:

- `labelStyle`: Specify the font style for the label displayed in the header. This attribute accepts CSS style attributes such as `font-size` or `color`.

For the complete list of CSS attributes, visit the World Wide Web Consortium's web site at:

<http://www.w3.org/TR/CSS21/>

- `titleHalign`: Specify the horizontal alignment of the header's title. By default, this attribute is set to `start` which aligns the title to the left in left-to-right mode and to the right in right-to-left mode. You can set this to `center` which aligns the title to the center or to `end` which aligns the title to the right in left-to-right mode and to the left in right-to-left mode.
- `useNodeColor`: Set this to `on` to have the header use the node color of the parent node.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap to your page. For more information, see [How to Add a Treemap to a Page](#).
- If you did not bind the treemap to a data control when you added the component to the page, add data to the treemap. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).

To configure a treemap node header:

1. In the Structure window, right-click the **dvt:treemapNode** node and choose **Insert Inside Treemap Node > Treemap Node Header**.
2. Right-click the **dvt:treemapNodeHeader** node and choose **Go to Properties**.
3. In the Properties window, enter a value for the following attributes:
 - **LabelStyle**: Enter the style for the node header title.
For example, to change the size of the title to 14 pixels and the color to white, enter the following value for **LabelStyle**:

```
font-size:14px;color: #FFFFFF
```
 - **TitleHalign**: Use the attribute's dropdown menu to change the default alignment to `center` or `end`. By default, this attribute is set to `start` which aligns the title to the left in left-to-right mode or to the right in right-to-left mode.
 - **UseNodeColor**: Use the attribute's dropdown menu to change the default to `on`.

What You May Need to Know About Treemap Node Headers

When you choose to use the node color in the header, the node color used is the color that would have been displayed in the treemap if that node was the bottom level of the treemap.

If your treemap is using the same color scheme across all hierarchical levels, then using the node color in the header can provide useful information. However, if you have specified a different color scheme for different levels of the hierarchy, using the node color may not make sense.

How to Customize Treemap Group Gaps

Customize the group gaps displayed between nodes by setting a value for the `groupGaps` attribute.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap to your page. For more information, see [How to Add a Treemap to a Page](#).

To customize treemap group gap display:

1. In the Structure window, right-click the **dvt:treemap** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section.
3. Use the **GroupGaps** dropdown menu to select a value for the group gap display. Valid values are:
 - `outer` (default): Gaps are displayed between the outer group nodes.
 - `all`: Gaps are displayed between all group nodes.
 - `none`: No gaps are displayed between group nodes.

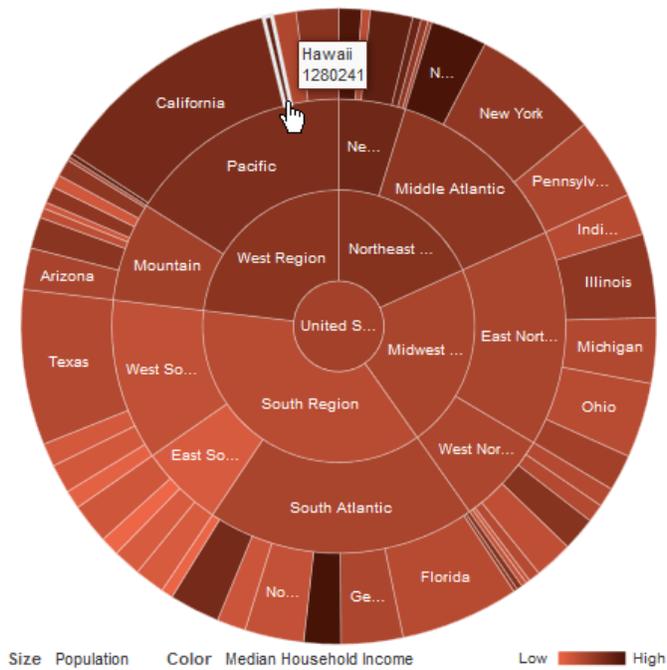
Adding Interactive Features to Treemaps and Sunbursts

ADF DVT Treemaps and Sunbursts have a number of interactive features to enhance user experience, such as tooltips, popups, selection support, context menus, and drilling. Treemaps also provide support for isolation of group nodes.

Configuring Treemap and Sunburst Tooltips

Define tooltips by specifying a value for the **dvt:treemapNode** or **dvt:sunburstNode** `shortDesc` attribute. You can specify simple text in this attribute, or you can specify an EL expression that pulls data from the treemap or sunburst and displays the additional detail about the node.

[Figure 32-35](#) shows a sunburst displaying the name and size of one of the sunburst nodes.

Figure 32-35 Sunburst Tooltip

To configure the tooltip to display detail about the node's label and size data, reference the `label` and `size` attributes in an EL expression. The EL expression pulls data from the managed bean that references the methods for setting the `label` and `size` attributes.

For example, to specify the values for the label and size attributes in the United States census example, enter the following for the `shortDesc` attribute in JDeveloper:

```
#{row.text}<br/>#{row.size}
```

Configuring Treemap and Sunburst Popups

Define popups in treemaps or sunbursts using the `af:popup` and `af:showPopupBehavior` tags.

Using the `af:popup` component with treemap and sunburst components, you can configure functionality to allow your end users to show and hide information in secondary windows, input additional data, or invoke functionality such as a context menu. See [Configuring Treemap and Sunburst Context Menus](#) to see how to display a context menu using the `af:popup` component.

How to Add Popups to Treemap and Sunburst Components

With ADF Faces components, JavaScript is not needed to show or hide popups. The `af:showPopupBehavior` tag provides a declarative solution, so that you do not have to

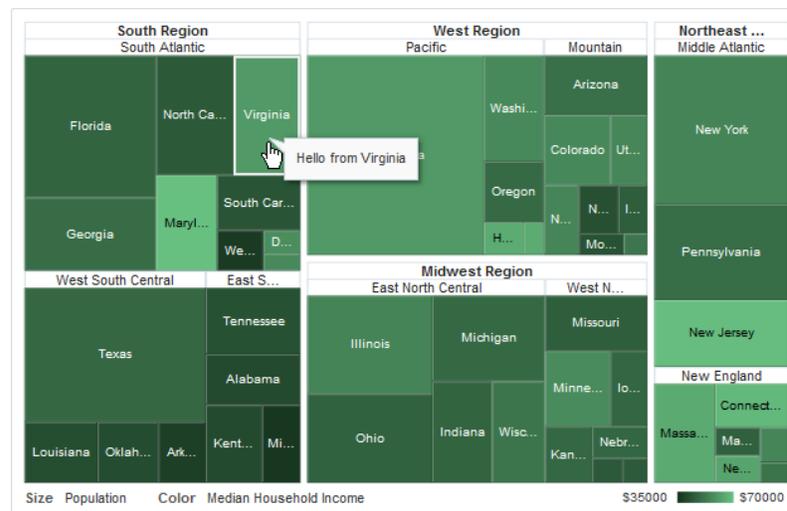
write JavaScript to open a popup component or register a script with the popup component.

This section provides an example for configuring a sunburst or treemap component to display popups using the `af:showPopupBehavior` tag.

To configure a popup using the `af:showPopupBehavior` and `af:popup` tags, define the `af:popup` component and associated methods, insert the `af:showPopupBehavior` tag as a child of the `dvt:treemapNode` or `dvt:sunburstNode` component and configure the `af:showPopupBehavior` component's tags for the trigger type and reference to the `af:popup` component's id attribute.

Figure 32-36 shows a treemap configured to display a brief message and the name of the treemap node as the user hovers the mouse over the treemap.

Figure 32-36 Treemap Showing Popup on Mouse Hover



The example below shows the code on the page to declare the popup.

```
<af:group id="g1">
  <af:outputText value="Hover on a node to show a popup."
    inlineStyle="font-size:medium;" id="ot1"/>
  <af:panelGroupLayout layout="horizontal" id="pgl1">
    <dvt:treemap id="treemap" value="#{treemap.censusData}" var="row"
      colorLabel="Median Household Income"
      sizeLabel="Population"
      displayLevelsChildren="3" legendSource="ag1"
      inlineStyle="width:700px; height:450px;"
      summary="Treemap Popup">
      <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}">
        <af:showPopupBehavior popupId="::noteWindowPopup"
          triggerType="mouseHover"/>
        <dvt:attributeGroups id="ag1" value="#{row.income}"
          type="color"
          attributeType="continuous"
          minValue="35000" maxValue="70000"
          minLabel="$35000" maxLabel="$70000">

```

```

        <f:attribute name="color1" value="#14301C" />
        <f:attribute name="color2" value="#68C182" />
    </dvt:attributeGroups>
</dvt:treemapNode>
</dvt:treemap>
</af:panelGroupLayout>
<af:popup childCreation="deferred" autoCancel="disabled"
    id="noteWindowPopup" launcherVar="source"
    eventContext="launcher" clientComponent="true"
    contentDelivery="lazyUncached">
    <af:setPropertyListener from="#{source.currentRowData.text}"
        to="#{treemap.noteWindowMessage}"
        type="popupFetch" />
    <af:noteWindow id="nw1">
        <af:outputFormatted value="Hello from #{treemap.noteWindowMessage}"
            id="of8" />
    </af:noteWindow>
</af:popup>
</af:group>

```

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).
- Add the ADF Faces `popup` component to your page and insert the menu, dialog, or window that you want the popup to display.

For example, the popup in [Figure 32-36](#) uses a note window to display the "Hello from Texas" message. To use this example, insert the ADF Faces `noteWindow` component inside the `popup` component, and insert the ADF Faces `outputFormatted` component inside the note window. The sample code is displayed in the code example above.

The example popup also includes the ADF Faces `setListener` component that retrieves the data from the treemap for use by the note window. In this example, the data is retrieved from the text attribute of the current node (`source.currentRowData.text`) and then stored in the `noteWindowMessage` string

variable in the treemap managed bean. To use this example, add the code shown in the example below to the treemap bean:

```
private String noteWindowMessage = null;

public void setNoteWindowMessage(String noteWindowMessage) {
    this.noteWindowMessage = noteWindowMessage;
}
public String getNoteWindowMessage() {
    return noteWindowMessage;
}
```

If you need help with managed beans, see [Creating and Using Managed Beans](#). For additional details about using popup windows to display dialogs, menus, and windows, see [Using Popup Dialogs, Menus, and Windows](#).

- Create any additional components needed to display the selection.

For example, the page in [Figure 32-36](#) uses an `af:outputText` component to prompt the user to hover on a node to show a popup. For additional information about configuring `af:outputText` components, see [Displaying Output Text and Formatted Output Text](#).

To add a popup to a treemap or sunburst:

1. In the Structure window, right-click the **dvt:treemapNode** or **dvt:sunburstNode** node and choose **Insert Inside Component Node > Show Popup Behavior**.

For example, to add the popup to a treemap, right-click the **dvt:treemapNode** node and choose **Insert Inside Treemap Node > Show Popup Behavior**.

2. Right-click the **af:showPopupBehavior** node and choose **Go to Properties**.
3. In the Properties window, enter a value for the following attributes:
 - **TriggerType**: Enter a value for the actions that will trigger the popup. Valid values are `click` and `mouseHover`.
 - **PopupId**: Reference the id of the popup component. You can enter the id directly or use the attribute's dropdown menu to choose **Edit** and select the id in the Edit Property: PopupId dialog.

For example, to reference the popup in the census data example, enter the following value for the **PopupId**:

```
::noteWindowPopup
```

What You May Need to Know About Adding Popups to Treemaps and Sunburst Components

Treemaps and sunbursts currently support only the `click` and `mouseHover` trigger types.

Popups do not display on the **Other** node. For additional information, see [What You May Need to Know About Configuring the Treemap and Sunburst Other Node](#).

Configuring Treemap and Sunburst Selection Support

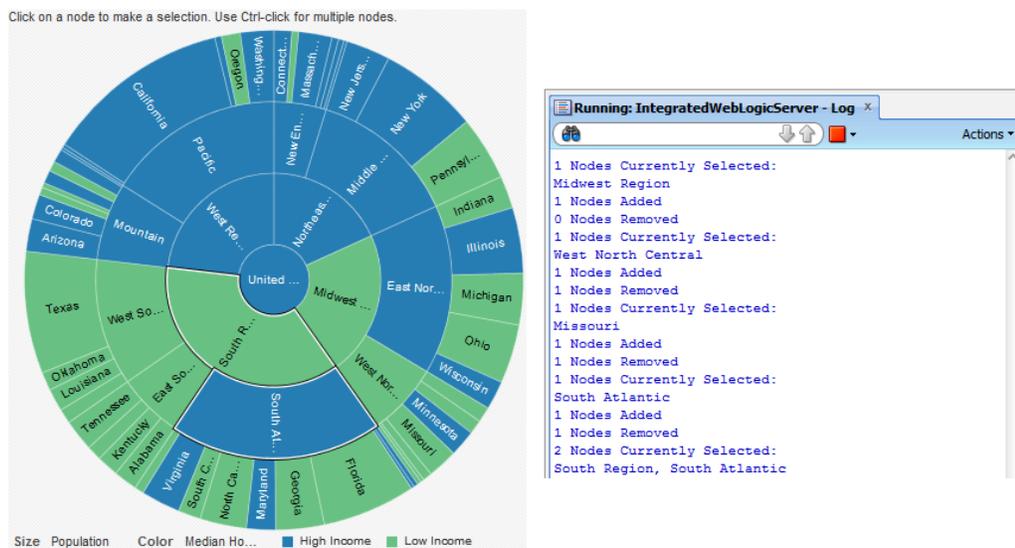
The treemap and sunburst components support single or multiple node selection. If the component allows multiple selections, users can select multiple nodes using a Control+click operation.

How to Add Selection Support to Treemap and Sunburst Components

When a user selects or deselects a node, the treemap or sunburst component invokes a `selectionEvent` event. You can register a custom `selectionListener` instance that can do post-processing on the treemap or sunburst component based on the selected node or nodes.

Figure 32-37 shows a simple example of a sunburst configured to use a custom selection listener. As the user makes single or multiple selections, the console displays the name of the node or nodes selected and the number of nodes added or removed from the selection.

Figure 32-37 Sunburst Illustrating Custom Selection Listener



The example below shows the `selectionListener` method used to respond to the user clicks and generate the output to the console. Store this method in the sunburst's managed or backing bean.

```
import javax.faces.component.UIComponent;
import oracle.adf.view.faces.bi.component.sunburst.UISunburst;
import org.apache.myfaces.trinidad.event.SelectionEvent;
import org.apache.myfaces.trinidad.model.RowKeySet;

public void selectionListener(SelectionEvent event) {
    UIComponent component = event.getComponent();
    if(component instanceof UISunburst) {
        UISunburst sunburst = (UISunburst) component;
        StringBuilder s = new StringBuilder();
```

```

        // Get the selected row keys and print
        RowKeySet selectedRowKeys = sunburst.getSelectedRowKeys();
        System.out.println(selectedRowKeys.size() + " Nodes Currently
Selected:");
        if (selectedRowKeys != null) {
            for (Object rowKey : selectedRowKeys) {
                TreeNode rowData = (TreeNode)sunburst.getRowData (rowKey);
                s.append (rowData.getText()).append(", ");
            }
            if (s.length() > 0)
                s.setLength (s.length() - 2);
            System.out.println(s);
        }
        // Get the row keys that were just added to the selection
        RowKeySet addedRowKeys = event.getAddedSet();
        System.out.println(addedRowKeys.size() + " Nodes Added");
        // Get the row keys that were just removed from the selection
        RowKeySet removedRowKeys = event.getRemovedSet();
        System.out.println(removedRowKeys.size() + " Nodes Removed");
    }
}

```

You declare the selection listener method in the treemap or sunburst component's `selectionListener` attribute and add any additional components to display the selection to the JSF page. In the example in this section, the listener is simply displaying the output to the console, and only the prompt to the user to make the selection is added to the page. The example below shows the portion of the page used to set up the sunburst. The `selectionListener` attribute is highlighted in bold font.

```

<af:panelGroupLayout id="pgl12">
    <af:group id="g5">
        <af:outputText value="Click on a node to make a selection. Use Ctrl-
click for multiple nodes."
            inlineStyle="font-size:large;" id="ot3"/>
        <dvt:sunburst id="s1" summary="SampleSunburst"
value="#{sunburst.censusData}"
            var="row" varStatus="rowStatus" displayLevelsChildren="3"
            colorLabel="Median Household Income"
sizeLabel="Population"
            inlineStyle="width:500px;height:500px;"
            selectionListener="#{sunburst.selectionListener}">
            <dvt:sunburstNode id="sn1" value="#{row.size}" label="#{row.text}"
                shortDesc="#{row.text}&lt;br/>#{row.size}">
                <dvt:attributeGroups id="agl" type="color "
                    value="#{row.income > 50000}"
                    label="#{row.income > 50000 ? 'High Income' :
'Low Income'}"
                    minLabel="Low" maxLabel="High">
                </dvt:attributeGroups>
            </dvt:sunburstNode>
        </dvt:sunburst>
    </af:group>
</af:panelGroupLayout>

```

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).
- Create the method that will define the `selectionListener` and return the selection state and store it in the treemap or sunburst component's managed or backing bean.

To use the same census data example, copy the example code into a managed bean named `sunburst`. If you need help with managed beans, see [Creating and Using Managed Beans](#).

- Create any additional components needed to display the selection.

For example, the page in [Figure 32-37](#) uses an `af:outputText` component to prompt the user to click on a node to make a selection. For additional information about configuring `af:outputText` components, see [Displaying Output Text and Formatted Output Text](#).

To add selection support to a treemap or sunburst:

1. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section and set the following properties:
 - **NodeSelection**: Set to `single` to enable selection support for single nodes only. Multiple selection is enabled by default.
 - **SelectionListener**: Enter the name of the method to be called when the user clicks on the nodes.

For example, for a managed bean named `sunburst` and a method named `selectionListener`, enter the following in the **SelectionListener** field:
`#{sunburst.selectionListener}`.

What You May Need to Know About Adding Selection Support to Treemaps and Sunbursts

Because treemaps and sunbursts use the same data model as the Tree component, selection events are defined in the `org.apache.myfaces.trinidad.event.SelectionEvent` library.

For additional information about selection support in a tree model, see [What Happens at Runtime: Tree Component Events](#).

For additional information about event handling in JDeveloper, see [Handling Events](#).

Configuring Treemap and Sunburst Context Menus

You can configure both treemaps and sunbursts to display context menus when a user right-clicks a node.

How to Configure Treemap and Sunburst Context Menus

Define treemap and sunburst context menus using these context menu facets:

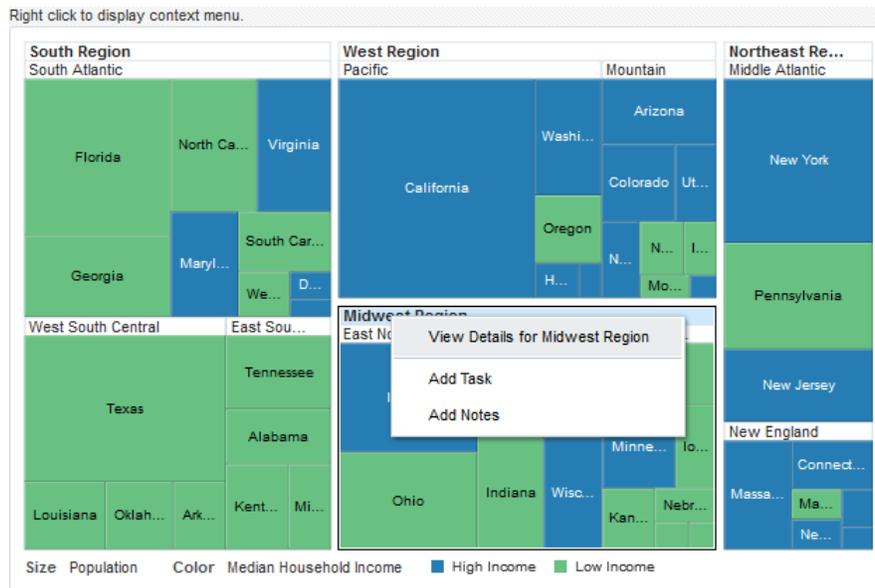
- `bodyContextMenu`: Specifies a context menu that is displayed on non-selectable elements in the treemap or sunburst component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the treemap or sunburst component.
- `multiSelectContextMenu`: Specifies a content menu that is displayed when multiple elements are selected in the treemap or sunburst component.

Each facet on a JSP or JSPX page supports a single child component. Facelets support multiple child components. For all of these facets to work, selection must be enabled in the treemap or sunburst's properties. Context menus are currently only supported in Flash.

You create a context menu by using `af:menu` components within an `af:popup` component. You can then invoke the context menu popup from another component, based on a specified trigger. For more information about configuring context menus, see [Using Popup Dialogs, Menus, and Windows](#).

[Figure 32-38](#) shows a sample treemap configured to display a context menu using the `contextMenu` facet when the user right-clicks on one of the treemap's regions, divisions, or nodes.

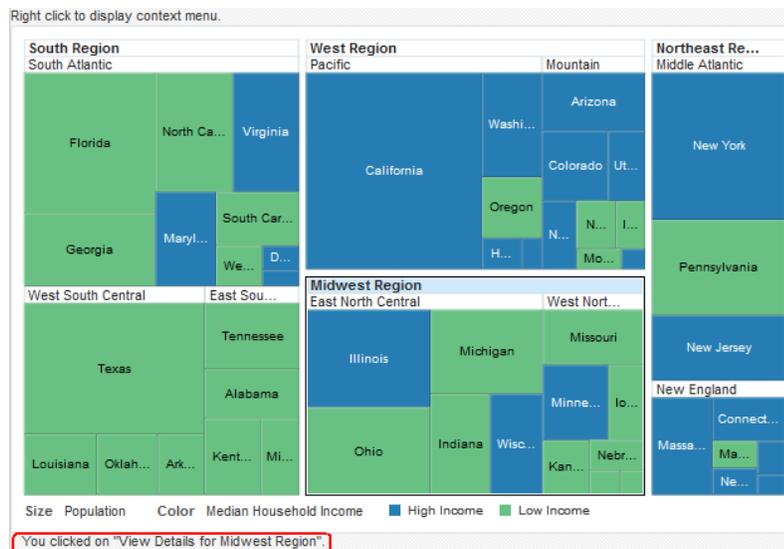
Figure 32-38 Treemap Context Menu



If the user selects **View Details for Midwest Region**, the application can provide additional information about the **Midwest Region** node.

Figure 32-39 shows the text output that is displayed below the treemap after the user chooses to view the details for the Midwest Region. In this example, the output simply verifies what the user clicked on, but this context menu could also be used to present additional details about the Midwest Region.

Figure 32-39 Context Menu Sample Output After Click



The example below shows the sample code used to configure the example treemap and the context menu.

```

<af:group id="g1">
  <af:outputFormatted value="Right click to display context menu."
id="of1"/>
  <dvt:treemap id="t1" displayLevelsChildren="3" summary="Sample Treemap"
    var="row" value="#{treemap.censusData}"
    varStatus="rowStatus"
    binding="#{treemapContextMenu.treemap}"
    colorLabel="Median Household Income"
    sizeLabel="Population"
    legendSource="ag1">
    <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}"
      shortDesc="#{row.text}<br/>Population: #{row.size}<br/
>Income: #{row.income}>
      <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
        label="#{row.income > 50000 ? 'High Income' :
'Low Income'}"
          type="color">
        </dvt:attributeGroups>
      </dvt:treemapNode>
    <f:facet name="contextMenu">
      <af:popup id="p1" contentDelivery="lazyUncached">
        <af:menu text="menu 1" id="m1">
          <af:commandMenuItem text="View Details for
#{treemapContextMenu.selectionState}"
            id="cm1"
              actionListener="#{treemapContextMenu.menuItemListener}"/>
          <af:group id="g2">
            <af:commandMenuItem text="Add Task" id="cmi2"
              actionListener="#{treemapContextMenu.menuItemListener}"/>
            <af:commandMenuItem text="Add Notes" id="cmi3"
              actionListener="#{treemapContextMenu.menuItemListener}"/>
          </af:group>
        </af:menu>
      </af:popup>
    </f:facet>
  </dvt:treemap>
  <af:spacer width="10" id="s1"/>
  <af:outputFormatted value="#{treemapContextMenu.status}" id="of2"
    clientComponent="true"
    binding="#{treemapContextMenu.outputFormatted}"/>
</af:group>

```

The example uses a backing bean named `treemapContextMenu` for the methods to set the treemap, return the selection state and respond to user clicks on the context menu. This example also uses the same classes and methods to set up the data for the

treemap as described in [Adding Data to Treemap and Sunburst Components](#). The example below shows the code for the `ContextMenuSample` class.

```
import javax.faces.component.UIComponent;
import javax.faces.event.ActionEvent;
import oracle.adf.view.faces.bi.component.treemap.UITreemap;
import oracle.adf.view.rich.component.rich.nav.RichCommandMenuItem;
import oracle.adf.view.rich.component.rich.output.RichOutputFormatted;
import org.apache.myfaces.trinidad.context.RequestContext;

public class ContextMenuSample {
    private UITreemap treemap;
    private String status;
    private RichOutputFormatted outputFormatted;
    public ContextMenuSample() {
    }
    public void setTreemap(UITreemap treemap) {
        this.treemap = treemap;
    }
    public UITreemap getTreemap() {
        return treemap;
    }
    public String getSelectionState() {
        if (treemap != null) {
            return TreemapSample.convertToString(treemap.getSelectedRowKeys(),
            treemap);
        } else
            return null;
    }
    public String getStatus() {
        return status;
    }
    public void setOutputFormatted(RichOutputFormatted outputFormatted) {
        this.outputFormatted = outputFormatted;
    }
    public RichOutputFormatted getOutputFormatted() {
        return outputFormatted;
    }
    /**
     * Called when a commandMenuItem is clicked. Updates the outputText
     with information about the menu item clicked.
     * @param actionEvent
     */
    public void menuItemListener(ActionEvent actionEvent) {
        UIComponent component = actionEvent.getComponent();
        if (component instanceof RichCommandMenuItem) {
            RichCommandMenuItem cmi = (RichCommandMenuItem)component;
            // Add the text of the item into the status message
            StringBuilder s = new StringBuilder();
            s.append("You clicked on \"").append(cmi.getText()).append("\".
            <br><br>");
            this.status = s.toString();
            // Update the status text component

RequestContext.getCurrentInstance().addPartialTarget(this.outputFormatted);
```

```

    }
  }
}

```

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).
- Create the managed bean that will define the `actionListener` and return the selection state.

To use the same census data example, copy the example code given above into a backing bean named `treemapContextMenu`. If you need help with managed beans, see [Creating and Using Managed Beans](#).

- Create any additional components needed to support the context menu.

For example, the page in [Figure 32-38](#) uses an `af:outputText` component to prompt the user to right-click to display a context menu. When the user selects the custom context menu item, the page uses an `af:outputFormatted` component to display a message confirming which node the user selected.

See the code sample given above for the details needed to configure the additional components. For additional information about `af:outputText` and `af:outputFormatted` components, see [Displaying Output Text and Formatted Output Text](#).

To add a context menu to a treemap or sunburst:

1. If your application is using a backing bean, do the following:
 - a. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
 - b. Expand the **Advanced** section and enter a value for the **Binding** attribute to associate the treemap with the managed bean that contains the methods for the context menu. Alternatively, choose **Edit** from the attribute's dropdown menu to create or select an existing bean and method.

The binding attribute is needed for the census data example because it includes the code to set up the treemap, but it also uses the data and methods from the same classes and methods that were described in [How to Add Data to Treemap or Sunburst Components](#). For example, for a backing bean named

treemapContextMenu, enter the following in the **Binding** field:
#{treemapContextMenu.treemap}.

2. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Insert Inside Treemap** or **Insert Inside Sunburst > Facet**.
3. In the Insert Facet dialog, enter the name of the facet that corresponds to the type of context menu that you wish to create.

For example, to define a `contextMenu` facet, enter the following in the **Name** field:
`contextMenu`.

4. Click **OK**.
The facet is created as a child of the **dvt:treemap** or **dvt:sunburst** node.
5. In the Structure window, right-click the **f:facet - context menu** node and choose **Insert Inside Facet > ADF Faces > Popup**.
6. Right-click the **af:popup** node and choose **Go to Properties**.
7. In the Properties window, set the following properties:
 - **ContentDelivery**: Set this to `LazyUncached`.
 - **AutoCancel**: Set this to `<default> enabled`.
 - **ChildCreation**: Set this to `<default> immediate`.
8. In the Structure window, right-click the **af:popup** node and choose **Insert Inside Popup > Menu**.
9. In the Structure window, right-click the **af:menu** node and choose **Insert Inside Menu > Menu Item** to create a menu item.
10. Right-click the **af:commandMenuItem** and choose **Go to Properties**.
11. In the Properties window, expand the **Common** section and set the following properties:

- **Text**: Enter the text to display in the menu.

For example, to duplicate the treemap census data example, enter the following in the **Text** field: `View Details for`
#{treemapContextMenu.selectionState}.

- **ActionListener**: Enter the name of the method to be called when the user selects the menu item.

For example, for a managed bean named `treemapContextMenu` and a method named `menuItemListener`, enter the following in the **ActionListener** field:
#{treemapContextMenu.menuItemListener}.

12. Repeat Step 9 through Step 11 for each menu item that you want the context menu to display.

 **Tip:**

To group related menu items, wrap the ADF Faces `af:group` component around the `af:commandMenuItem` as shown in the code sample above for the Context Menu class. For information about the `af:group` component, see [Grouping Related Items](#).

13. To configure additional context menu facets, repeat Step 2 through Step 12.

What You May Need to Know About Configuring Treemap and Sunburst Context Menus

Due to technical limitations when using the Flash rendering format, context menu contents are currently displayed using the Flash Player's context menu.

This imposes several limitations defined by the Flash Player:

- Flash does not allow for submenus in its context menu.
- Flash limits custom menu items to 15. Any built-in menu items for the component, for example, a pie graph `interactiveSliceBehavior` menu item, will count towards the limit.
- Flash limits menu items to text-only. Icons or other controls possible in ADF Faces menus are not possible in Flash menus.
- Each menu caption must contain at least one visible character. Control characters, new lines, and other white space characters are ignored. No caption can be more than 100 characters long.
- Menu captions that are identical to another custom item are ignored, whether the matching item is visible or not. Menu captions are compared to built-in captions or existing custom captions without regard to case, punctuation, or white space.
- The following captions are not allowed, although the words may be used in conjunction with other words to form a custom caption: **Save, Zoom In, Zoom Out, 100%, Show All, Quality, Play, Loop, Rewind, Forward, Back, Movie not loaded, About, Print, Show Redraw Regions, Debugger, Undo, Cut, Copy, Paste, Delete, Select All, Open, Open in new window, and Copy link.**
- None of the following words can appear in a custom caption on their own or in conjunction with other words: **Adobe, Macromedia, Flash Player, or Settings.**

Additionally, since the request from Flash for context menu items is a synchronous call, a server request to evaluate EL is not possible when the context menu is invoked. To provide context menus that vary by selected object, the menus will be pre-fetched if the context menu popup uses the setting `contentDelivery="lazyUncached"`. For context menus that may vary by state, this means that any EL expressions within the menu definition will be called repeatedly at render time, with different selection and currency states. When using these context menus that are pre-fetched, the application must be aware of the following:

- Long running or slow code should not be executed in any EL expression that may be used to determine how the context menu is displayed. This does not apply to `af:commandMenuItem` attributes that are called after a menu item is selected, such as `actionListener`.
- In the future, if the Flash limitations are solved, the ADF context menu may be displayed in place of the Flash context menu. To ensure upgrade compatibility, you should code such that an EL expression will function both in cases where the menu is pre-fetched, and also where the EL expression is evaluated when the menu is invoked. The only component state that applications should rely on are selection and currency.

Configuring Treemap and Sunburst Drilling Support

Drilling support enables the user to navigate through the treemap or sunburst hierarchy by clicking the component's group headers or by double-clicking the individual nodes.

How to Configure Treemap and Sunburst Drilling Support

Enable drilling support through the treemap or sunburst node's `drilling` attribute.

JDeveloper includes the necessary code to support drilling. However, you may want the application to perform some other task when the node is drilled. You can define a method to perform the additional task and add it as a drill listener to the treemap's or sunburst's managed or backing bean.

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).
- If you wish to add a drill listener, create the method that will define the listener and add it to the treemap's managed or backing bean.

For more information about handling events, see [Handling Events](#). If you need help with beans, see [Creating and Using Managed Beans](#).

To add drilling support to a treemap or sunburst

1. In the Structure window, right-click the `dvt:treemapNode` or `dvt:sunburstNode` node and choose **Go to Properties**.
2. In the Properties window, expand the **Advanced** section, and use the **Drilling** attribute's dropdown list to set the **Drilling** attribute to one of the following values:
 - `replace`: allows the user to double-click a node to set it as the new root of the treemap or sunburst
 - `insert` (sunburst only): allows the user to expand or collapse the children of a node
 - `insertAndReplace` (sunburst only): allows the user to double-click a node to set it as the root of the hierarchy and allows the user to expand or collapse the children of a node

3. If your application includes a drill listener, do the following:
 - a. In the Structure window, right-click the **dvt:treemap** node and choose **Go to Properties**.
 - b. In the Properties window, expand the **Behavior** section.
 - c. From the **DrillListener** attribute's dropdown menu, choose **Edit**.
 - d. In the Edit Property dialog, use the search box to locate the treemap's managed bean.
 - e. Expand the managed bean node and select the method that contains the drill listener.
 - f. Click OK.

The expression is created.

For example, for a managed bean named `sampleTreemap` and a method named `sampleDrillListener`, the Expression Builder generates the `code#{sampleTreemap.sampleDrillListener}` as the value for the drill listener.

What You May Need to Know About Treemaps and Drilling Support

Drilling is recommended when there are additional layers of data that can be displayed. Unlike isolation, it is a server side operation that will fetch additional data from the tree model. To focus on group data that is already displayed, use the treemap isolate feature.

For more information, see [Configuring Isolation Support \(Treemap Only\)](#).

How to Add Drag and Drop to Treemaps and Sunbursts

You can configure treemaps and sunbursts as drag sources and drop targets for drag and drop operations between supported components on a page.

To add drag support to a treemap or sunburst, add the `af:dragSource` tag to the treemap and add the `af:dropTarget` tag to the component receiving the drag. The component receiving the drag must include the `org.apache.myfaces.trinidad.model.RowKeySet` data flavor as a child of the `af:dropTarget` and also define a `dropListener` method to respond to the drop event.

To add drop support to a treemap or sunburst, add the `af:dropTarget` tag to the treemap or sunburst and include the data flavors that the treemap or sunburst will support. Add a `dropListener` method to a treemap or sunburst managed bean that will respond to the drop event.

The following procedure shows how to set up a treemap or sunburst as a simple drag source or drop target. For more detailed information about configuring drag and drop on ADF Faces or ADF Data Visualization components, see [Adding Drag and Drop Functionality](#).

Before you begin:

It may be helpful to have an understanding of how treemap and sunburst attributes and child tags can affect functionality. For more information about configuring treemap attributes and child tags, see [Configuring Treemaps](#). For information about configuring sunburst attributes and child tags, see [Configuring Sunbursts](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

You will need to complete these tasks:

- Add a treemap or sunburst to your page. For more information, see [How to Add a Treemap to a Page](#) or [How to Add a Sunburst to a Page](#).
- If you did not bind the treemap or sunburst to a data control when you added the component to the page, add data to the treemap or sunburst. For information about adding data to treemaps or sunbursts using UI-first development, see [Adding Data to Treemap and Sunburst Components](#).
- Create any additional components needed to support the drag and drop.

For example, the page in [Figure 32-18](#) uses an `af:outputText` component to prompt the user to drag a treemap node to the indicated text. When the user drags a node to the text, the page uses an `af:outputFormatted` component to display a message confirming which node the user dragged.

The example below shows the sample code for the completed page. For additional information about `af:outputText` and `af:outputFormatted` components, see [Displaying Output Text and Formatted Output Text](#).

```
<af:group id="g1">
  <af:panelGroupLayout id="pgl2" layout="horizontal">
    <af:outputText value="Drag Source Demo" inlineStyle="font-size:large;" id="ot2"/>
    <af:spacer width="10" height="10" id="s1"/>
    <af:outputText value="Drag a Treemap Node to the Text" id="ot1"/>
  </af:panelGroupLayout>
  <af:panelGroupLayout id="pgl3" layout="horizontal">
    <dvt:treemap id="t1" value="#{treemap.censusData}" var="row"
      displayLevelsChildren="3"
      colorLabel="Median Household Income"
      sizeLabel="Population" summary="Discrete
Treemap"
      legendSource="ag1">
      <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}"
        shortDesc="#{row.text}&lt;br/>Population: #{row.size}&lt;br/>
Income: #{row.income}">
        <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
          label="#{row.income > 50000 ? 'High Income' : 'Low
Income' }"
          type="color"/>
        </dvt:treemapNode>
      <af:dragSource defaultAction="COPY" actions="COPY MOVE LINK"/>
    </dvt:treemap>
    <af:spacer width="20" id="s2"/>
    <af:outputFormatted value="#{treemap.dropText}" id="of1">
      <af:dropTarget dropListener="#{treemap.fromDropListener}">
        <af:dataFlavor
flavorClass="org.apache.myfaces.trinidad.model.RowKeySet"/>
      </af:dropTarget>
    </af:outputFormatted>
  </af:panelGroupLayout>
</af:group>
```

```

</af:panelGroupLayout>
</af:group>

```

The example below shows the sample code for the page in [Figure 32-19](#). In this example, the treemap is configured as the drop target.

```

<af:group id="g1">
  <af:panelGroupLayout id="pgl4" layout="horizontal">
    <af:outputText value="Drop Target Demo" inlineStyle="font-size:large;"/>
    <af:spacer width="10" id="s2"/>
    <af:outputText value="Drag From the Text to the Treemap" id="ot1"/>
  </af:panelGroupLayout>
  <af:panelGroupLayout id="pgl3" layout="horizontal">
    <dvt:treemap id="t1" value="#{treemap.censusData}" var="row"
      displayLevelsChildren="3"
      colorLabel="Median Household Income"
      sizeLabel="Population" summary="Discrete
Treemap"
      legendSource="ag1">
      <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}"
        shortDesc="#{row.text}&lt;br/>Population: #{row.size}&lt;br/>
>Income: #{row.income}">
        <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
          label="#{row.income > 50000 ? 'High Income' : 'Low
Income'}"
          type="color"/>
        </dvt:treemapNode>
      <af:dropTarget dropListener="#{treemap.toDropListener}"
        actions="MOVE COPY LINK">
        <af:dataFlavor flavorClass="java.lang.Object"/>
      </af:dropTarget>
    </dvt:treemap>
    <af:spacer width="20" id="s1"/>
    <af:outputFormatted value="#{treemap.dragText}" id="of1"
      clientComponent="true">
      <af:componentDragSource/>
    </af:outputFormatted>
  </af:panelGroupLayout>
</af:group>

```

To add drag and drop support to a treemap or sunburst:

1. To configure a treemap or sunburst as a drop target, in the Components window, from the Operations panel, drag a **Drop Target** and drop it as a child to the treemap or sunburst component.
2. In the Insert Drop Target dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the treemap's or sunburst's managed bean. Alternatively, use the dropdown menu to choose **Expression Builder** and enter an EL Expression for the drop listener.

For example, to add a method named `toDropListener()` on a managed bean named `treemap`, choose **Edit**, select **treemap** from the dropdown menu, and click **New** on the right of the **Method** field to create the `toDropListener()` method.

The example below shows the sample drop listener and supporting methods for the treemap displayed in [Figure 32-19](#).

```
// imports needed by methods
import java.util.Map;
import oracle.adf.view.rich.dnd.DnDAction;
import oracle.adf.view.rich.event.DropEvent;
import oracle.adf.view.rich.datatransfer.DataFlavor;
import oracle.adf.view.rich.datatransfer.Transferable;
import org.apache.myfaces.trinidad.context.RequestContext;
import org.apache.myfaces.trinidad.render.ClientRowKeyManager;
import javax.faces.context.FacesContext;
import oracle.adf.view.faces.bi.component.treemap.UITreemap;
import javax.faces.component.UIComponent;
// variables need by methods
private String dragText = "Drag this text onto a node";
// drop listener
public DnDAction toDropListener(DropEvent event) {
    Transferable transferable = event.getTransferable();
    DataFlavor<Object> dataFlavor =
DataFlavor.getDataFlavor(Object.class);
    Object transferableObj = transferable.getData(dataFlavor);
    if(transferableObj == null)
        return DnDAction.NONE;
    // Build up the string that reports the drop information
    StringBuilder sb = new StringBuilder();
    // Start with the proposed action
    sb.append("Drag Operation: ");
    DnDAction proposedAction = event.getProposedAction();
    if(proposedAction == DnDAction.COPY) {
        sb.append("Copy<br>");
    }
    else if(proposedAction == DnDAction.LINK) {
        sb.append("Link<br>");
    }
    else if(proposedAction == DnDAction.MOVE) {
        sb.append("Move<br>");
    }
    // Then add the rowKeys of the nodes that were dragged
    UIComponent dropComponent = event.getDropComponent();
    Object dropSite = event.getDropSite();
    if(dropSite instanceof Map) {
        String clientRowKey = (String) ((Map) dropSite).get("clientRowKey");
        Object rowKey = getRowKey(dropComponent, clientRowKey);
        if(rowKey != null) {
            sb.append("Drop Site: ");
            sb.append(getLabel(dropComponent, rowKey));
        }
    }
    // Update the output text
    this.dragText = sb.toString();

RequestContext.getCurrentInstance().addPartialTarget(event.getDragComponent());
    return event.getProposedAction();
}
```

```

    }

    public String getDragText() {
        return dragText;
    }

    private String getLabel(UIComponent component, Object rowKey) {
        if(component instanceof UITreemap) {
            UITreemap treemap = (UITreemap) component;
            TreeNode rowData = (TreeNode) treemap.getRowData(rowKey);
            return rowData.getText();
        }
        return null;
    }

    private Object getRowKey(UIComponent component, String clientRowKey) {
        if(component instanceof UITreemap) {
            UITreemap treemap = (UITreemap) component;
            ClientRowKeyManager crkm = treemap.getClientRowKeyManager();
            return crkm.getRowKey(FacesContext.getCurrentInstance(), component,
            clientRowKey);
        }
        return null;
    }
}

```

3. Click OK to enter the Insert Data Flavor dialog.
4. In the Insert Data Flavor dialog, enter the object that the drop target will accept. Alternatively, use the dropdown menu to navigate through the object hierarchies and choose the desired object.

For example, to allow the `af:outputFormatted` component to drag text to the treemap, enter `java.lang.Object` in the Insert Data Flavor dialog.

5. In the Structure window, right-click the **af:dropTarget** node and choose **Go to Properties**.
6. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

7. To use the treemap or sunburst as the drop target, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the component that will be the source of the drag.

For example, drag and drop a **Drag Source** as a child to an `af:outputFormatted` component.
 - b. In the Properties window, in the component's **Value** field, reference the public variable that you created in the drop listener for the treemap or sunburst in Step 2.

For example, for a drop listener named `toDropListener()` and a variable named `dropText`, enter the following in the component's **Value** field:

```
#{treemap.dropText}
```

8. To configure the treemap or sunburst as a drag source, in the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the treemap or sunburst.
9. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: COPY, MOVE, or LINK.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

10. To specify the default action that the drag source will support, use the **DefaultAction** attribute's dropdown menu to choose COPY, MOVE, or LINK.

The treemap in the drag and drop example in [Figure 32-18](#) uses COPY as the default action.

11. To make another component the drop target for drags from the treemap or sunburst, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drop Target** onto the component that will receive the drop.

For example, the page in the drag and drop example in [Figure 32-19](#) contains an `af:outputFormatted` component that displays the results of the drop.

- b. In the Insert Drop Target dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the appropriate managed bean. Alternatively, use the dropdown menu to choose **Expression Builder** and enter an EL Expression for the drop listener.

For example, to add a method named `fromDropListener()` on a managed bean named `treemap`, choose **Edit**, select `treemap` from the dropdown menu, and click **New** on the right of the **Method** field to create the `fromDropListener()` method.

The example below shows the sample drop listener for the treemap displayed in [Figure 32-18](#). This example uses the same imports and helper methods used in the drop listener example above, and they are not included here.

```
// Additional import needed for listener
import org.apache.myfaces.trinidad.model.RowKeySet;
// Variables needed by method
private String dropText = "Drop a node here";
// Drop listener
public DnDAction fromDropListener(DropEvent event) {
    Transferable transferable = event.getTransferable();
    DataFlavor<RowKeySet> dataFlavor =
DataFlavor.getDataFlavor(RowKeySet.class);
    RowKeySet rowKeySet = transferable.getData(dataFlavor);
    if(rowKeySet == null || rowKeySet.getSize() <= 0)
        return DnDAction.NONE;
    // Build up the string that reports the drop information
    StringBuilder sb = new StringBuilder();
    // Start with the proposed action
    sb.append("Drag Operation: ");
    DnDAction proposedAction = event.getProposedAction();
    if(proposedAction == DnDAction.COPY) {
        sb.append("Copy<br>");
    }
}
```

```

    }
    else if(proposedAction == DnDAction.LINK) {
        sb.append("Link<br>");
    }
    else if(proposedAction == DnDAction.MOVE) {
        sb.append("Move<br>");
    }
    // Then add the rowKeys of the nodes that were dragged
    sb.append("Nodes: ");
    UIComponent dragComponent = event.getDragComponent();
    for(Object rowKey : rowKeySet) {
        sb.append(getLabel(dragComponent, rowKey));
        sb.append(", ");
    }
    // Remove the trailing ,
    sb.setLength(sb.length()-2);
    // Update the output text
    this.dropText = sb.toString();

    RequestContext.getCurrentInstance().addPartialTarget(event.getDropComponent());
    return event.getProposedAction();
}

```

c. Click OK to enter the Insert Data Flavor dialog.

d. In the Insert Data Flavor dialog, enter `org.apache.myfaces.trinidad.model.RowKeySet`.

For example, to allow the `af:outputFormatted` component to drag text to the treemap, enter `org.apache.myfaces.trinidad.model.RowKeySet` in the Insert Data Flavor dialog.

e. In the Structure window, right-click the **af:dropTarget** node and choose **Go to Properties**.

f. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.

For example, enter the following in the **Actions** field to allow all operations:

```
COPY MOVE LINK
```

g. In the component's **Value** field, reference the public variable that you created in the drop listener for the treemap or sunburst in Step 2.

For example, for a drop listener named `fromDropListener()` and a variable named `dragText`, enter the following in the component's **Value** field:

```
#{treemap.dragText}
```

Configuring Isolation Support (Treemap Only)

Isolation allows the user to click a group header to maximize the display of the group's data. The isolation feature is enabled by default when the group header is displayed.

How to Disable Isolation Support

If you wish to disable isolation, set the `Isolate` attribute of the **dvt:treemapNodeHeader** node to `off`.

Before you begin:

It may be helpful to have an understanding of how treemap attributes and treemap child tags can affect functionality. For more information, see [Configuring Treemaps](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Treemap and Sunburst Components](#).

Add a treemap to your page. For more information, see [How to Add a Treemap to a Page](#).

Add treemap node headers to your treemap. For more information, see [How to Configure Treemap Node Headers](#).

To disable isolation support on a treemap group:

1. In the Structure window, expand the **dvt:treemapNode** node.
2. Right-click the **dvt:treemapNodeHeader** node and choose **Go to Properties**.
3. In the Properties window, expand the **Advanced** section.
4. From the **Isolate** attribute's dropdown menu, choose `off`.
5. If your treemap has multiple nodes, repeat Step 1 through Step 4 to disable isolation support for each of the nodes.

What You May Need to Know About Treemaps and Isolation Support

Isolation is a client-side operation that allows the user to focus on data that is already displayed. If your treemap has multiple child levels and you want the user to access levels that are not already displayed, use drilling instead.

To add drilling support, see [How to Configure Treemap and Sunburst Drilling Support](#).

Using Diagram Components

This chapter describes how to use the ADF Data Visualization `diagram` component to display data in diagrams using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of the component.

If your application uses the Fusion technology stack, then you can also use data controls to create diagrams. For more information, see the "Creating Databound Diagrams" section in the *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About the Diagram Component](#)
- [Using the Diagram Component](#)
- [Using the Diagram Layout Framework](#)

About the Diagram Component

ADF DVT Diagrams are flexible and highly configurable components that can display a wide range of data items. Use diagrams to model, represent, and visualize information using a shape called a node to represent data, and links to represent relationships between nodes.

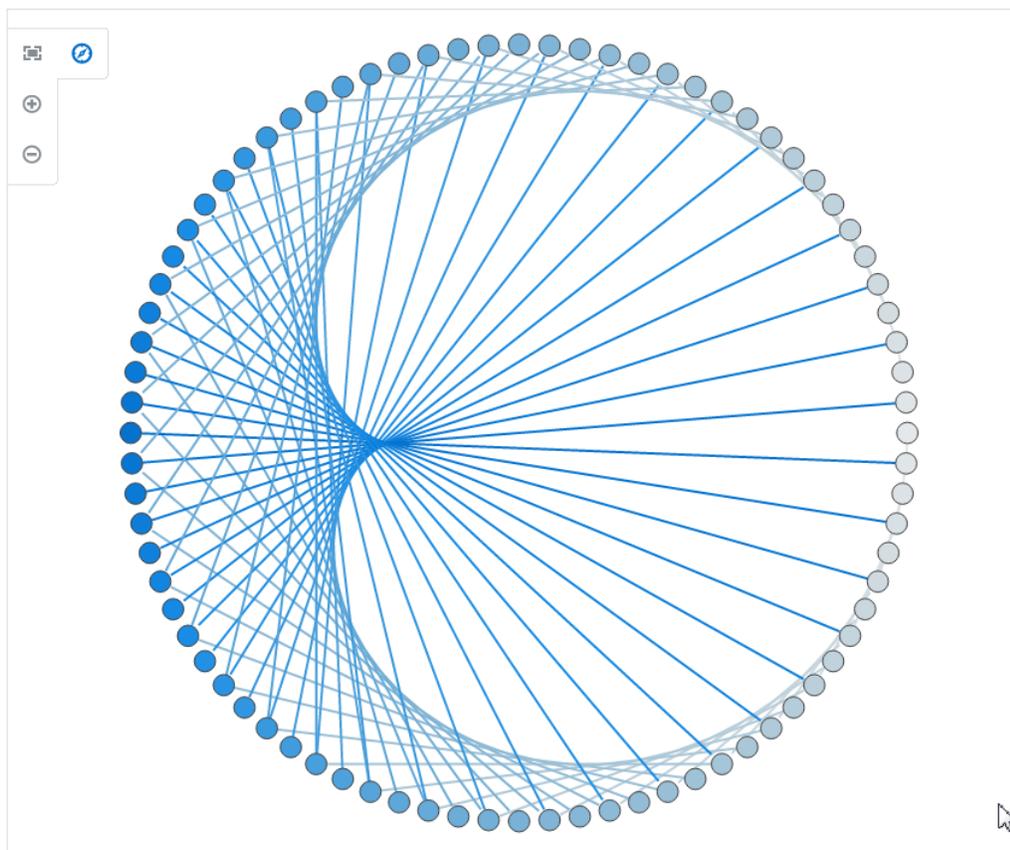
Diagrams can represent basic data using simple shapes or images with a label, or more complex data using zoom levels and stylistic attribute values such as colors for each unique group in the data set. The layout of diagram links and nodes is configured through a framework using one or more custom JavaScript files. The framework supports rendering in multiple platforms including SVG and Java2D.

For example, diagrams can model a database schema to logically group objects such as tables, views, and stored procedures; represent an employee tree with a complex set of information for each employee at different zoom levels; or visualize the net migration between US states using a sunburst layout.

Diagram Use Cases and Examples

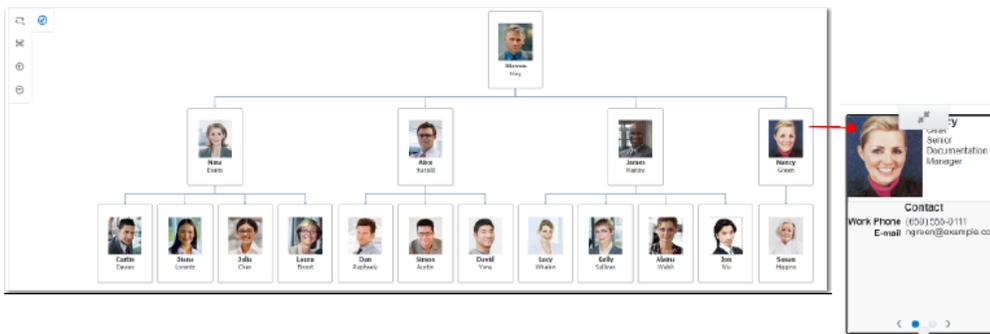
Diagrams use shapes for the node, and lines for links to represent the relationships between the nodes. [Figure 33-1](#) shows a simple diagram configured in a circular layout pattern with circles and lines representing the relationships between them. The example includes the default control panel for diagram zooming.

Figure 33-1 Simple Diagram with Nodes and Links with Control Panel



Diagrams can also be configured to visually display hierarchical data with a master-detail relationship. [Figure 33-2](#) shows an employee tree diagram at runtime that includes a control panel, a number of nodes, and links that connect the nodes. Also illustrated is a node panel card that uses `af:showDetailItem` elements to display multiple sets of data at different zoom levels.

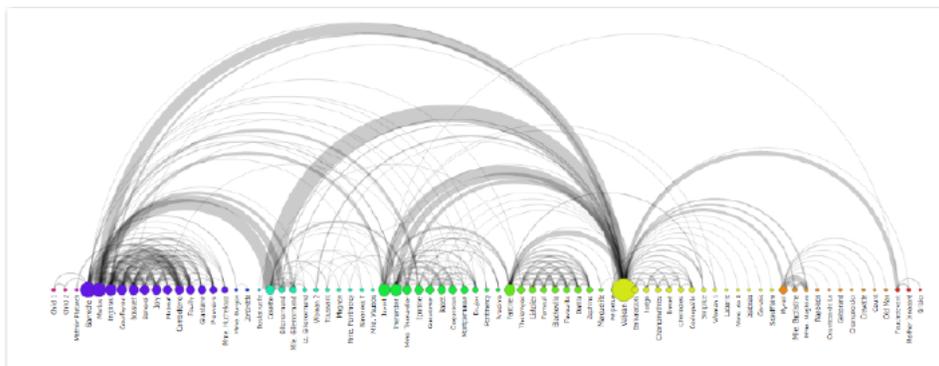
Figure 33-2 Employee Tree Diagram with Control Panel



The diagram component can also be configured to display an arc diagram, a graphical display to visualize graphs in a one-dimensional layout. Nodes are displayed along a single axis, while representing the edges or connections between nodes with arcs.

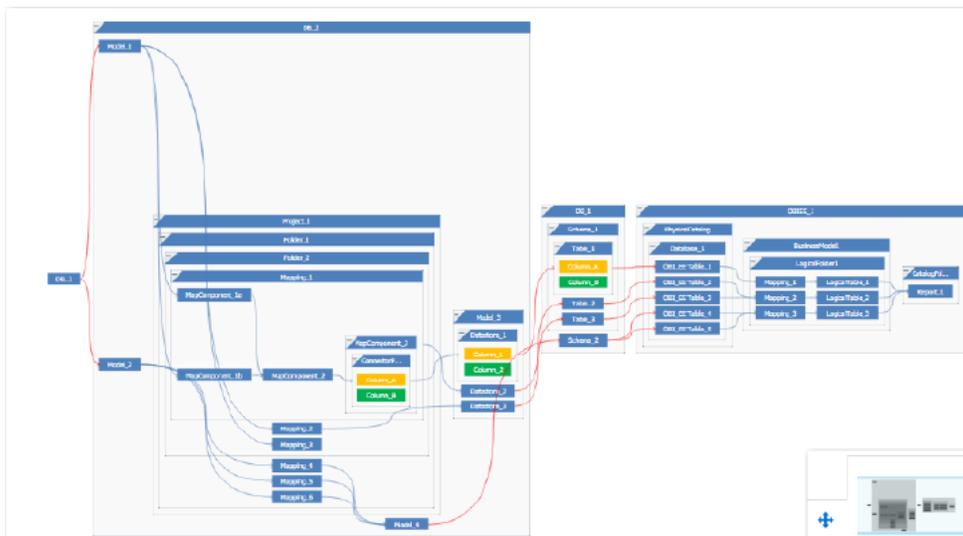
Figure 33-3 shows an arc diagram that use characters from Victor Hugo's "Les Miserables" novel to display co-appearances between any pair of characters that appear in the same chapter of the book.

Figure 33-3 Arc Diagram Displaying Character Co-Apearances



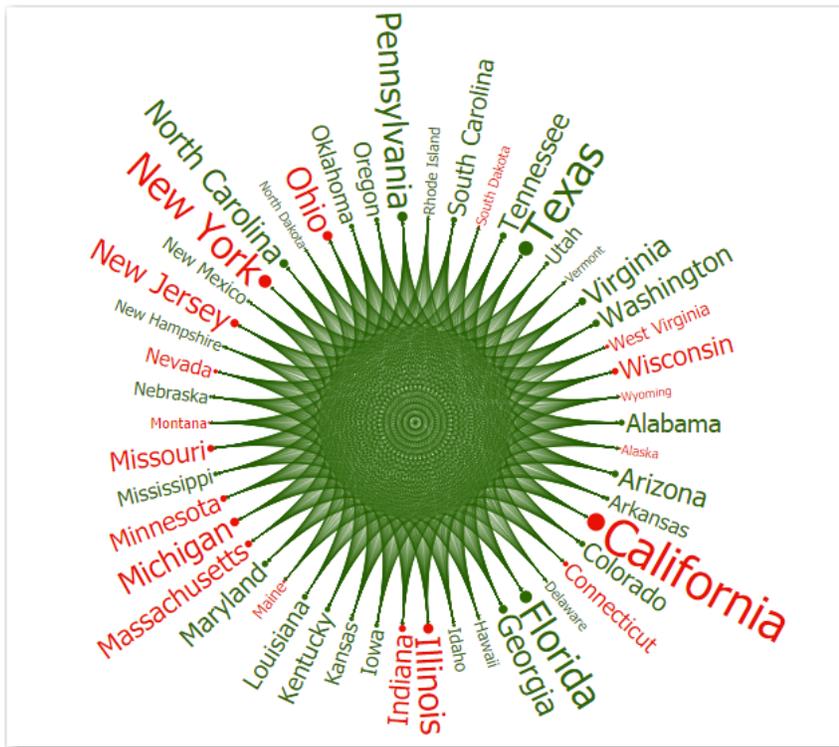
Diagrams can be configured to display a database schema. Figure 33-4 shows a database schema layout diagram with an overview window, a simplified view of the diagram that provides the user with context and navigation options.

Figure 33-4 Database Schema Layout Diagram with Overview



Diagrams can also be configured in a sunburst layout to display quantitative hierarchical data across two dimensions, represented visually by size and color. Figure 33-5 shows a diagram displaying US state to state migration data using attribute groups to display net results.

Figure 33-5 Diagram Displaying State to State Net Migration Data



End User and Presentation Features of Diagrams

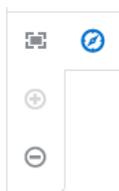
The ADF Data Visualization `diagram` component provides a range of features for end users, such as zooming, grouping, and panning. They also provide a range of presentation features, such as legend display, and customizable colors and label styles.

To use and customize diagram components, it may be helpful to understand these features and components:

- **Control Panel:** Provides tools for a user to manipulate the position and appearance of a diagram component at runtime. By default, it appears in a hidden state in the upper left-hand corner of the diagram. Users click the **Hide or Show Control Panel** button to hide or expand the control panel.

Figure 33-6 shows the control panel expanded to display the zoom controls and zoom-to-fit option.

Figure 33-6 Diagram Control Panel



- **Zooming:** Enabled by default, users can change the view of the diagram in an expanded or collapsed display. The maximum and minimum scale of the zoom can be configured for the diagram.
- **Panning:** Enabled by default, users can select the diagram in any open area and move the entire diagram right, left, up, and down to reposition the view.

 **Note:**

The `PanDirection` attribute allows you to decide whether to constrain panning to horizontal or vertical only.

- **LayoutBehaviour:** Lets users control whether the diagram calls out to the application to perform layout at each individual container level or just once globally.
- **Popup Support:** Diagram components can be configured to display popup dialogs, windows, and menus that provide information when the user clicks or hovers the mouse over a node. [Figure 33-7](#) shows a node popup window when the user mouses over the node.

Figure 33-7 Diagram Node Popup



- **Legend Support:** Diagrams display legends to provide a visual clue to the type of data represented by the shapes and color. If the component uses attribute groups to specify colors based on conditions, the legend can also display the colors used and indicate what value each color represents.
- **Overview Window:** Displays a simplified view of the diagram that provides the user with context and navigation options. The Overview Window viewport can be dragged to pan the diagram. Users click the **Hide or Show Overview Window** button to hide or expand the window. [Figure 33-8](#) shows the overview window expanded and panned to display a portion of a database schema layout.

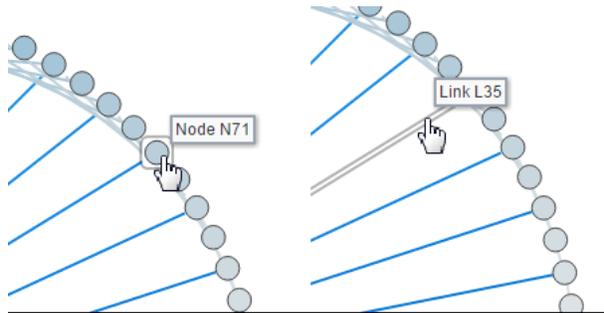
Figure 33-8 Overview Window



- **Context Menus:** Diagrams support the ability to display context menus to provide additional information about the selected node.
- **Attribute Groups:** Diagram nodes support the use of the `dvt:attributeGroups` tag to generate stylistic attribute values such as colors for each unique group in the data set.

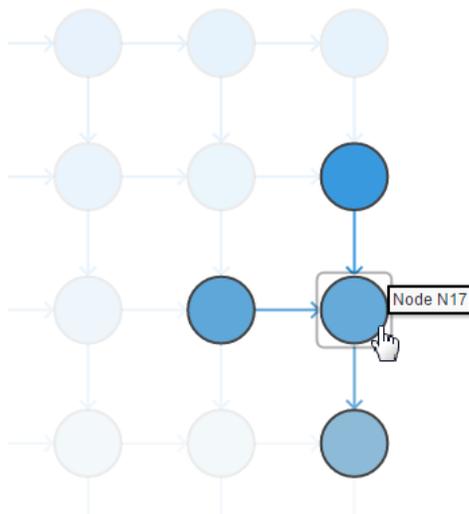
- **Selection Support:** Diagrams support the ability to respond to user clicks on one or more nodes or links to display information about the selected element.
- **Tooltip Support:** Diagrams support the ability to display additional information about a node or link when the user moves the mouse over the element. [Figure 33-9](#) shows node and link tooltips.

Figure 33-9 Node and Link Tooltips



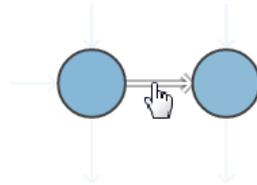
- **Drilling Support:** Diagram components support drilling to navigate through a hierarchy and display more detailed information about a node.
- **Highlighting Support:** Diagrams can be configured to highlight the view of nodes and links when the user hovers the mouse over an element in the diagram. Highlighting can be specified for these values: `none`, `node`, `nodeAndIncomingLinks`, `nodeAndOutgoingLinks`, and `nodeAndLinks`. [Figure 33-10](#) shows the highlighted view of a node upon hover in a diagram when the `nodeAndLinks` value is specified.

Figure 33-10 Node Highlighting Configuration



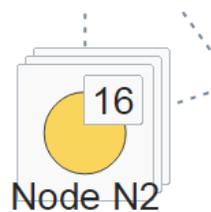
[Figure 33-11](#) shows the highlighted view of a link upon hover in a diagram when the `nodeAndLinks` value is specified.

Figure 33-11 Link Highlighting Configuration



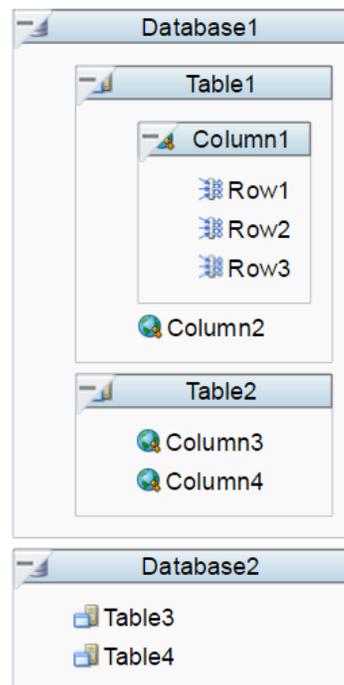
- **Stacking:** You can group nodes and display the nodes in a stacked display. [Figure 33-12](#) shows a stack of nodes. By default, the number of nodes in the stack are displayed.

Figure 33-12 Grouped Nodes Displayed as a Stack



- **Container Nodes:** Nodes can represent a hierarchical relationship using one or more container nodes. [Figure 33-13](#) shows a set of container nodes in a database schema.

Figure 33-13 Diagram Container Nodes in a Database Schema



- **Node Controls:** A tooltip describing the control is displayed by default.

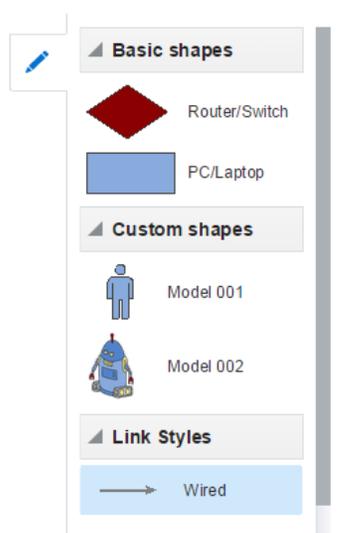
Table 33-1 Diagram Node Controls

Control	Name	Description
	Isolate	Allows user to detach a single node for isolated display in the diagram.
	Preview Stack	Allows user to display all the nodes in a group.
	Unstack	Allows user to display a grouped set of nodes so that all nodes are visible.
	Restack	Allows user to collapse a set of associated nodes into a grouped display.
	Restore	If you configured a panel card, displays the list of <code>af:showDetailItem</code> elements that you have defined. Users can use the panel selector to show the same panel on all nodes at once.
	Close	Allows user to display all the nodes in a group.
	Drill	Allows user to navigate through the node's hierarchy and display additional detail about a node.

Table 33-1 (Cont.) Diagram Node Controls

Control	Name	Description
	Additional Links	Allows user to display the links and nodes associated with a selected node.

- **Palette:** Diagrams support a panel of node and link items that can be used for drag and drop into a layout configuration. Users click the **Hide or Show Palette** button to hide or expand the window. [Figure 33-14](#) shows a palette of nodes and links available for creating a diagram.

Figure 33-14 Palette of Diagram Nodes and Links

- **Display in Printable or Emailable Pages:** ADF Faces allows you to output your JSF page from an ADF Faces web application in a simplified mode for printing or emailing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scrollbars and buttons. If a page is to be emailed, the page must be simplified so that email clients can correctly display it. For information about creating simplified pages for these outputs, see [Using Different Output Modes](#).

Additional Functionality for Diagram Components

You may find it helpful to understand other ADF Faces features before you implement your diagram component. Additionally, once you have added a diagram component to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that diagram components can use:

- You may want a diagram to refresh a node to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: If enabled, users can change the way the diagram displays at runtime, and those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: Diagram components are accessible, as long as you follow the accessibility guidelines for the component. See [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Skins and styles: You can customize the appearance of diagram components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Content Delivery: You can configure your diagram to fetch data from the data source immediately upon rendering the components, or on a second request after the components have been rendered using the `contentDelivery` attribute. For more information, see [Content Delivery](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound diagrams based on how your ADF Business Components are configured. JDeveloper provides a wizard for data binding and configuring your diagram. For more information, see the "Creating Databound Diagram Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), image formats, and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Diagram Component

To use the ADF DVT Diagram component, add the diagram component to a page using the Component Palette window. Then define the data for the diagram and

specify the JavaScript method to invoke the diagram layout. Complete any additional configuration in JDeveloper using the tag attributes in the Properties window.

Diagram Data Requirements

A diagram represents two sets of separate, but related data objects; a set of nodes, and a set of links that join those nodes. Data is supplied as two separate collections of data provided either as an implementation of the `List` interface (`java.util.ArrayList`), or a `CollectionModel` (`org.apache.myfaces.trinidad.model.CollectionModel`).

The data can be of any type, typically `String`, `int`, or `long`.

Both collections of diagram nodes and links require an attribute that represents the unique Id for each row in the collection. The collections are mapped using a `value` attribute to stamp out each instance of the node or link using a component to iterate through the collection. Each time a child component is stamped out, the data for the current component is copied into a `var` property used by the data layer component in an EL Expression. Once the diagram has completed rendering, the `var` property is removed, or reverted back to its previous value. By convention, `var` is set to `node` or `link`.

The values for the `value` attribute must be stored in the node's or link's data model or in classes and managed beans if you are using UI-first development.

Note:

The `CollectionModel` includes data provided by the ADF binding layer in the form of a table or hierarchical binding, which can be used to data bind from the ADF model using data controls.

Configuring Diagrams

The `diagram` component has configurable attributes and child components that you can add or modify to customize the display or behavior of the diagram.

The prefix `dvt:` occurs at the beginning of each diagram component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

You can configure diagram child components, attributes, and supported facets in the following areas:

- **Diagram (`dvt:diagram`):** Wraps the collection of nodes (`diagramNodes`) and links (`diagramLinks`), and specifies the layout (`clientLayout`) reference for the diagram. Configure the following attributes to control the diagram display:
 - **Animation:** Use the `animationOnDisplay` attribute to control the initial animation. To change the animation duration from the default duration of 500 milliseconds, modify the `animationDuration` attribute.
 - **Empty text (`emptyText`):** Use the `emptyText` attribute to specify the text to display if there are no nodes to display.

- Zoom (`zooming`): By default the diagram can be displayed in an expanded or compressed view. Use `maxZoom` and `minZoom` to specify the size of the scale of the zoom.
- Pan (`panning`): By default with a device specific gesture, users can select and move the diagram up, down, right, or left.
- Client layout (`clientLayout`): Used to register a JavaScript method invoked to perform layout for a diagram. For detailed information, see [Using the Diagram Layout Framework](#).
- Diagram nodes (`diagramNodes`): Wraps the collection of diagram nodes. Configure the following attributes to manage the collection of diagram nodes:
 - `value`: Specify the data model for the node collection. Can be an instance of `javafx.faces.CollectionModel`.
 - `var`: Name of the EL variable used to reference each element of the node collection.
 - `varStatus`: Use to provide contextual information about the state of the component.
 - `groupBy`: Use to stack nodes by listing the attribute group IDs in a space-separated list.
 - `highlightBehavior`: Use to specify the highlight behavior when the user hovers the mouse over a node in the diagram. Valid values are: `none`, `node`, `nodeAndIncomingLinks`, `nodeAndOutgoingLinks`, and `nodeAndLinks`.
- Node (`diagramNode`): Used to stamp out each instance of the node in the (`diagramNodes`) collection. A node is a shape that references the data in a diagram, for example, employees in an organization or computers in a network.

The node component supports the use of these `f:facet` elements:

- Zoom: Supports the use of one or more `f:facet` elements that display content at different zoom levels, with more detail content available at the higher zoom levels.
 - * `f:facet name="zoom100"`
 - * `f:facet name="zoom75"`
 - * `f:facet name="zoom50"`
 - * `f:facet name="zoom25"`

The `f:facet` element supports the use of many ADF Faces components as child components, such as `af:outputText`, `af:image`, and `af:panelGroupLayout`, in addition to the ADF Data Visualization `marker` and `panelCard` components.

At runtime, the node contains controls that allow users to navigate between nodes and to show or hide other nodes by default. For information about specifying node content and defining zoom levels, see

- Label: Provides the ability to specify a label that can be independently positioned by the layout.
- Background: Use to specify a background for the zoom facets, automatically scaled to fit the node size. Content is stamped on top of the background. The facet supports `af:image` and `dvt:marker`.

- **Overlay:** Use to position one or more markers overlaid on the zoom facets. The overlay is positioned using a `dvt:pointLocation` element for a `dvt:marker`. Use an `af:group` element when specifying more than one overlay marker.
- **Container:** Templates provided for stamping out areas of a parent (container) node. Available for specifying top, bottom, right, left, and left-to-right reading direction right and left areas.
- **Attribute groups (`attributeGroups`):** Use this optional child tag of a diagram node child element, typically the `dvt:marker` component, to generate style values for each unique value, or group, in the data set. You can also apply attribute groups to more than one component, for example, in all defined zoom facets to maintain consistency while zooming, or to set the background color of an `af:panelGroupLayout`. For more information, see

Attribute groups are necessary to provide information for the display of the diagram legend and are therefore recommended. For more information, see

- **Links (`diagramLinks`):** Wraps the collection of diagram links. Configure the following attributes to manage the collection of diagram links:
 - **value:** Specify the data model for the link collection. Can be an instance of `javax.faces.CollectionModel`.
 - **var:** Name of the EL variable used to reference each element of the link collection.
 - **varStatus:** Use to provide contextual information about the state of the component.
 - **highlightBehavior:** Use to specify the highlight behavior when the user hovers the mouse over a link in the diagram. Valid values are: `none` and `link`.
- **Link (`diagramLink`):** Use to stamp out each instance of the link in the (`diagramLinks`) collection. Links connect one node with another node. You can configure the display of links using the following attributes:
 - **linkColor:** Use to specify the color of the link using a CSS named color.
 - **linkStyle:** Use to specify the appearance of the link. Valid values are `solid` (default), `dash`, `dot`, `dashDot`.
 - **linkWidth:** Use to specify the width in pixels for the link. The default value is 1.
 - **startConnectorType:** When required, use to specify one of the available images to use for the starting node link connector. Valid values are `none`, `arrowOpen`, `arrow`, `arrowConcave`, `circle`, `rectangle`, and `rectangleRounded`.
 - **endConnectorType:** When required, use to specify one of the available images to use for the terminal node link connector. Valid values are `none`, `arrowOpen`, `arrow`, `arrowConcave`, `circle`, `rectangle`, and `rectangleRounded`.
 - **label:** use to specify the label to use for the link.

The link component supports the use of these `f:facet` elements:

- **startConnector:** Supports the use of a custom image for the starting node link connector.
- **endConnector:** Supports the use of a custom image for the terminal node link connector.

- `label`: Supports the use of a custom label for the link.

For information about how to customize the appearance of the link and add labels, see

- `Legend` (`legend`): Use to display multiple sections of marker and label pairs. Define the legend as a child of the diagram component.
- `Palette` (`palette`): Use to create a panel for creating nodes and links elements (`paletteItem`). The palette can display multiple sections of node and label pairs, supporting custom title headers and declarative control over the ordering of such sections. The palette can contain multiple sections (`paletteSection`) of nodes and a single section of links (`paletteLinkDefs`).

What You May Need to Know About Using the Default Diagram Layout

When you create a diagram using UI-first development in JDeveloper, a Create Diagram wizard provides the option of using a default client layout that you can use or edit to specify the layout of your diagram. The default layout is based on force-directed graph drawing algorithms.

To view the complete code for the default layout, you can view the file in JDeveloper, or see [Code Sample for Default Client Layout](#).

When you create a diagram using the default client layout, JDeveloper performs the following actions:

- Registers the layout with the ADF runtime environment as a JavaScript (JS) feature. The creation process automatically adds a JS file in the `ViewController/src/js/layout` directory with the following naming convention:

```
ApplicationNameDiagramLayout.js
```

For example, if your application is named `DiagramSample`, the JS file is named `DiagramSampleDiagramLayout.js`.

- Creates the `adf-js-features.xml` file if it doesn't already exist and adds the `ApplicationNameDiagramLayout` as the feature-name and the path to the JS file as the feature-class. The file is placed in the `ViewController/src/META-INF` directory.

The following example shows the `adf-js-features.xml` file for an application named `DiagramSample`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<features xmlns="http://xmlns.oracle.com/adf/faces/feature">
  <feature>
    <feature-name>DiagramSampleDiagramLayout</feature-name>
    <feature-class>js/layout/DiagramSampleDiagramLayout.js</feature-class>
  </feature>
</features>
```

- References the JavaScript file from the diagram using the `dvt:clientLayout` element.

The following example shows the sample code referencing the JavaScript file:

```
<dvt:diagram layout="DiagramSampleDiagramLayout" id="d2"
  <dvt:clientLayout method="DiagramSampleDiagramLayout.forceDirectedLayout"
    featureName="DiagramSampleDiagramLayout"
    name="DiagramSampleDiagramLayout"/>
```

```
...
</diagram>
```

- Configures the development environment to copy the .js file type from the source path to the output directory when invoking the Make or Build command.

When you design and use a custom client layout for your diagram, you must complete these actions manually. For more information, see [How to Register a Custom Layout](#).

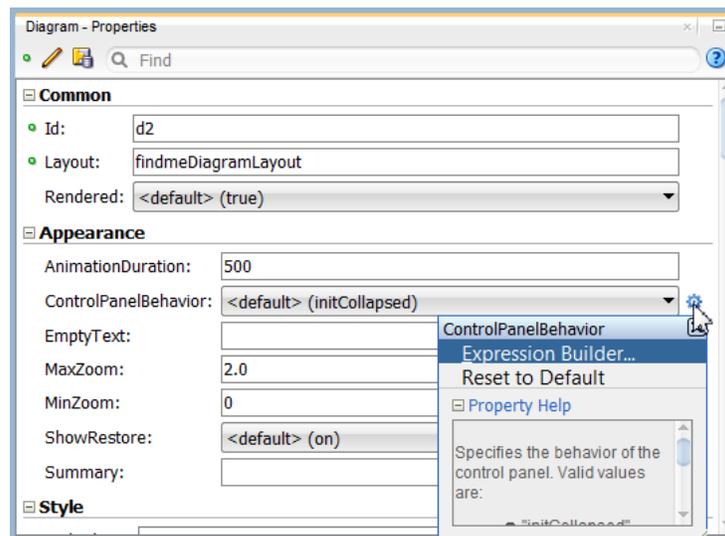
How to Add a Diagram to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a diagram to a JSF page. When you drag and drop a diagram component onto the page, a Create Diagram wizard displays.

If you click **Finish**, the diagram is added to your page, and you can use the Properties window to specify data values and configure additional display attributes. Alternatively, you can choose to bind the data during creation and use the Configure Diagram wizard - Node and Link Data page to configure the associated node and link data.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. [Figure 33-15](#) shows the dropdown menu for a dialog `layout` attribute.

Figure 33-15 Diagram Layout Attribute Dropdown Menu



 **Note:**

If your application uses the Fusion technology stack, then you can use data controls to create a diagram and the binding will be done for you. For more information, see the "Creating Databound Diagram Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how diagram attributes and child tags can affect functionality. For more information, see [Configuring Diagrams](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Diagram Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a diagram to a page:

1. In the ADF Data Visualization page of the Components window, from the Common panel, drag and drop a **Diagram** onto the page to open the Create Diagram wizard.
2. In the Client Layout page, choose one of the following options:
 - **Default Client Layout:** Use the layout available in JDeveloper. The layout is based on force-directed graph drawing algorithms.
 - **Custom Client Layout:** Use a custom designed client layout.
 - **No Layout:** Create the diagram without specifying a layout for the nodes and links.

If you select the default layout or specify a custom layout, you can open the file in JDeveloper. You can also click **Edit Component Definition** in the Properties window to display the Edit Diagram - Configure Client Layout dialog and then click **Search** to display the method in the popup.

3. Click **Finish** to add the diagram to the page.

 **Note:**

If the JSP page is run, only the control panel will be displayed since the diagram is not bound to data at this point.

Optionally, click **Next** to use the wizard to bind the diagram on the Node and Link Data page by navigating to the ADF data control that represents the data you wish to display on the diagram nodes and links. If you choose this option, the data binding fields in the dialog will be available for editing. For help with the dialog, press F1 or click **Help**.

4. In the Properties window, view the attributes for the diagram. Use the help button to display the complete tag documentation for the `diagram` component.
5. Expand the **Appearance** section, and enter values for the following attributes:
 - **ControlPanelBehavior:** Use the attribute's dropdown menu to change the default display of the control panel from `initCollapsed` to `initExpanded` or `Hidden`.
 - **Summary:** Enter a summary of the diagram's purpose and structure for screen reader support.

What Happens When You Add a Diagram to a Page

JDeveloper generates a set of tags when you drag and drop a diagram from the Components window onto a JSF page and choose not to bind the data during creation.

The generated code is:

```
<dvt:diagram layout="DiagramSampleDiagramLayout" id="d2">
  <dvt:clientLayout method="DiagramSampleDiagramLayout.forceDirectedLayout"
    featureName="DiagramSampleDiagramLayout"
    name="DiagramSampleDiagramLayout" />
  <dvt:diagramNodes id="dn1">
    <dvt:diagramNode id="dn2">
      <f:facet name="zoom100">
        <dvt:marker id="m1" />
      </f:facet>
    </dvt:diagramNode>
  </dvt:diagramNodes>
  <dvt:diagramLinks id="dl1">
    <dvt:diagramLink id="dl2" />
  </dvt:diagramLinks>
</dvt:diagram>
```

If you choose to bind the data to a data control when creating the diagram, JDeveloper generates code based on the data model. For more information, see the "Creating Databound Diagram Components" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

How to Create Diagram Nodes

When the diagram is added to your page, you can use the Configure Diagram wizard - Diagram Nodes page to configure nodes to the diagram.

If you click **Finish**, the default characteristics of a diagram node's marker are added to the diagram.

Before you begin:

It may be helpful to have an understanding of how diagram attributes and child tags can affect functionality. See [Configuring Diagrams](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. See [Additional Functionality for Diagram Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To create diagram nodes:

1. In the ADF Data Visualization page of the Components window, from the Common panel, drag and drop a **Diagram** onto the page to open the Create Diagram wizard
2. In the Client Layout page, choose one of the following options:
 - **Default Client Layout:** Use the layout available in JDeveloper. The layout is based on force-directed graph drawing algorithms.

- **Custom Client Layout:** Use a custom designed client layout.
 - **No Layout:** Create the diagram without specifying a layout for the nodes and links.
3. Click **Next** to use the wizard to bind the diagram on the Node and Link Data page by navigating to the ADF data control that represents the data you wish to display on the diagram nodes and links.
 4. In the Node and Link Data page, choose the appropriate value from the **Node Id** dropdown menu.
 5. Click **Next** to go to the Diagram Nodes page and configure the default display characteristics of a diagram node.
 6. Click **Finish** to add the node to the diagram.

 **Note:**

Optionally, if you need more control, you can specify attribute groups to apply display properties to markers based upon grouping rules

7. In the Properties window, view the attributes for the node. Use the help button to display the complete tag documentation for the `diagramNode` component.
8. Expand the different sections, and enter appropriate values.

 **Note:**

- In the **Other** section, the `DragFlavor` and `DropFlavors` attributes control which nodes can be dragged onto which drop targets.
- In the **Other** section, the `OverviewContainerStyle` and `OverviewStyles` attributes control how nodes are displayed in the overview. The `OverviewContainerStyle` attribute represents the style in the Overview Window when the node is disclosed and the `OverviewStyle` attribute represents the style when the node is not disclosed.

Using the Diagram Layout Framework

The ADF DVT Diagram layout framework supports the display of diagram nodes and links in whatever configuration meets the business needs of the diagram. Written in JavaScript and based on HTML5, the framework supports rendering in multiple platforms, including SVG and Java2D.

Diagrams require a client layout configuration defined in a JavaScript method that you add to the application as a feature. Client layouts specify how to lay out the nodes and links on a page. Diagrams can be configured to work with more than one layout, as in the case of parent and child nodes, but at least one layout must be registered with the application. All node and link display rendering is specified by the diagram component and all layout logic is contained in the layout JavaScript file.

The diagram component includes a default layout based on force-directed graph drawing algorithms that you can use and modify as needed. The default `forceDirectedLayout` positions the diagram nodes so that all the links are of more or less equal length, and there are as few crossing links as possible

You can also provide your own client layout configuration in a JavaScript object that you add as an application feature using the DVT Diagram APIs.

Layout Requirements and Processing

The basic process of laying out a diagram is to loop through an array of nodes and an array of links and set positioning information on each of them.

Custom diagram layout code uses the DVT diagram base classes defined in [Table 33-2](#).

Table 33-2 DVT Diagram Base Classes

Base Class	Description/Methods (partial)
<code>DvtDiagramLayoutContext</code>	<p>Defines the context for a layout call.</p> <p><i>layout:</i> <code>getLayout()</code>, <code>getLayoutAttributes()</code>, <code>getContainerId()</code>, <code>get/setContainerPadding()</code>, <code>getComponentSize()</code>, <code>get/setViewport()</code></p> <p><i>nodes:</i> <code>getNodeCount()</code>, <code>getNodeByIndex()</code>, <code>getNodeById()</code></p> <p><i>links:</i> <code>getLinkCount()</code>, <code>getLinkByIndex()</code>, <code>getLinkById()</code></p> <p><i>helpers:</i> <code>isLocaleR2L()</code>, <code>localToGlobal()</code></p>
<code>DvtDiagramLayoutContextLink</code>	<p>Defines the link context for a layout call.</p> <p><i>top level:</i> <code>getId()</code>, <code>get/setPoints()</code>, <code>getStartConnectorOffset()</code>, <code>getEndConnectorOffset()</code>, <code>getLinkWidth()</code>, <code>getLayoutAttributes()</code>, <code>isPromoted()</code></p> <p><i>start/end nodes:</i> <code>getStartId()</code>, <code>getEndId()</code></p> <p><i>labels:</i> <code>get/setLabelPosition()</code>, <code>getLabelBounds()</code>, <code>get/setLabelRotationAngle()</code>, <code>get/setLabelRotationPoint()</code></p>
<code>DvtDiagramLayoutContextNode</code>	<p>Defines the node context for a layout call.</p> <p><i>top level:</i> <code>getId()</code>, <code>getBounds()</code>, <code>getContentBounds()</code>, <code>get/setPosition()</code>, <code>getLayoutAttributes()</code>, <code>getContainerId()</code>, <code>get/setContainerPadding()</code></p> <p><i>state:</i> <code>isReadOnly()</code>, <code>isDisclosed()</code>, <code>getSelected()</code></p> <p><i>labels:</i> <code>get/setLabelPosition()</code>, <code>getLabelBounds()</code>, <code>get/setLabelRotationAngle()</code>, <code>get/setLabelRotationPoint()</code></p>
<code>DvtDiagramPoint</code>	Point class, used for function parameters and return values (x,y).
<code>DvtDiagramRectangle</code>	Rectangle, Dimension class, used for function parameters and return values (x,y,w,h).

You can view the entire JavaScript API for the DVT Diagram [here](#)

Configuring Diagram Layouts

The DVT `diagram` child `clientLayout` component identifies the JavaScript needed to run the layout of the nodes and links using an alias referenced by the diagram in the `layout` attribute. The DVT `clientLayout` component is responsible for laying out all Diagram nodes and links.

The DVT `diagram` component is unique in its flexibility, allowing users to build custom layouts. It does not contain pre-built layouts, hence giving users control over the layout and appearance via JavaScript functions. Diagrams can use multiple client layouts; for example, when you have a parent container node and a child node, but at least one layout must be registered with the application. When implementing a client layout, it is important to consider factors such as node and link interaction, zoom behavior, and locale (including BiDi locales).

The DVT `clientLayout` component has three attributes:

- `name`: Alias for the layout referenced from the diagram's `layout` property.
- `method`: Identifies the name of a JavaScript function that will perform the layout.
- `featureName`: Identifies the JavaScript source files required to locate and run the function identified in the `method` attribute.

How to Register a Custom Layout

When you create a custom layout for a diagram, you must configure the ADF Faces runtime environment by performing the following actions:

- Register the layout with the ADF runtime environment as a JavaScript (JS) feature.
- Create the `adf-js-features.xml` file if it doesn't already exist and add the `ApplicationNameDiagramLayout` as the feature-name and the path to the JS file as the feature-class.
- Reference the JavaScript file from the diagram using the `dvt:clientLayout` element.
- Configure the development environment to copy the `.js` file type from the source path to the output directory when invoking the Make or Build command.

Before you begin:

It may be helpful to have an understanding of how diagram attributes and child tags can affect functionality. For more information, see [Configuring Diagrams](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Diagram Components](#).

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To register a custom layout:

1. In the application `ViewController/src/js/layout` directory, add the custom JavaScript file using the following naming convention:

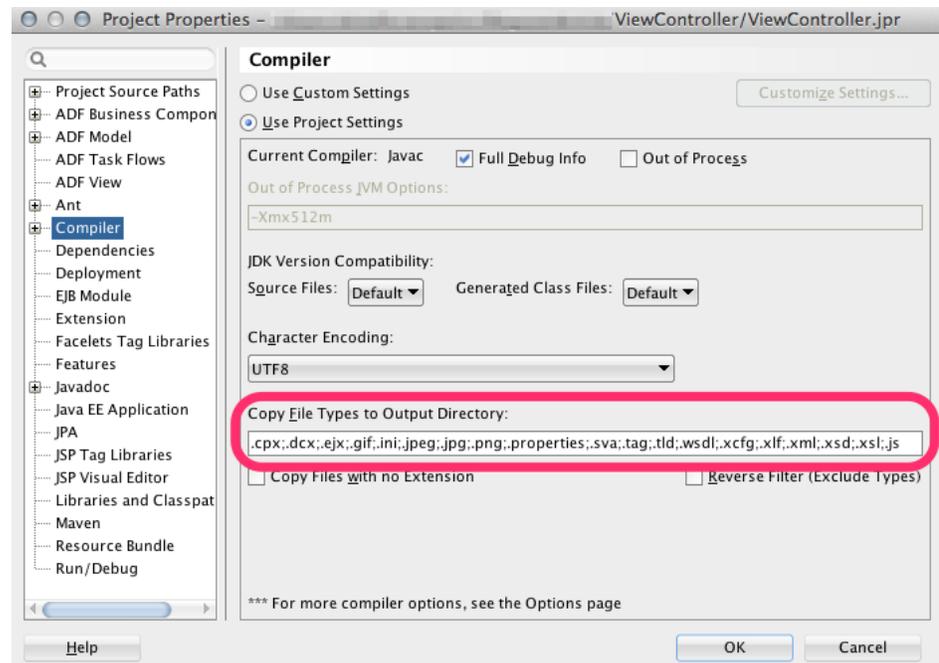
`ApplicationNameDiagramLayout.js`

For example, if your application is named `DiagramSample`, the JS file is named `DiagramSampleDiagramLayout.js`.

2. In the Applications Window > Projects Panel, right click the `ViewController/src/META-INF` folder and choose **New > XML file**.
3. In File Name enter `adf-js-features.xml` and click **OK**.
4. In the XML Editor define a feature name and associate the name with a JavaScript file. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<features xmlns="http://xmlns.oracle.com/adf/faces/feature">
  <feature>
    <feature-name>DiagramSampleDiagramLayouts</feature-name>
    <feature-class>js/layouts/DiagramSampleDiagram.js</feature-class>
  </feature>
</features>
```

5. Double-click the project in the Application Navigator to open the Project Properties dialog.
6. In the Compiler settings page > Copy File Types to Output Directory, add a `.js` file type to the semi-colon separated list. The completed dialog is shown below:



7. Click **OK**.

Designing Simple Client Layouts

Client layouts specify the layout of nodes and links in a diagram.

For example, the layout of diagram nodes in a simple vertical pattern is shown in [Figure 33-16](#).

Figure 33-16 Vertical Diagram Node Layout



In the layout, the layout context object passed to the layout function contains the information about the number of nodes using the `getNodeCount()` API, and provides access to each node object using the `getNodeByIndex()` API. After looping through the nodes to access each node, a call to `setPosition()` API is used to locate the node on the diagram space. The position is passed in the form of a `DvtDiagramPoint()` object which defines the simple x and y coordinates.

The JavaScript code for a simple vertical layout of nodes is here:

```

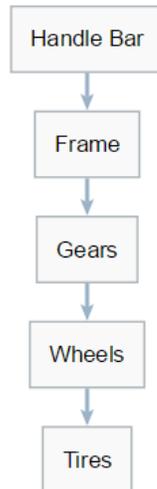
var DemoVerticalLayout = {
};

/**
 * Main function that does the layout (Layout entry point)
 * @param {DvtDiagramLayoutContext} layoutContext Object that defines a context for
layout call
 */
DemoVerticalLayout.layout = function (layoutContext) {
  var padding = 10; //default space between nodes, that can be overwritten by layout
attributes
  var currX = 0; //x coordinate for the node's center
  var currY = 0; //y coordinate for the node's top edge
  var nodeCount = layoutContext.getNodeCount();
  for (var ni = 0; ni < nodeCount; ni++) {
    var node = layoutContext.getNodeByIndex(ni);
    var bounds = node.getContentBounds();
    //Position the node - in order to do it find a position for the node's top left
corner
    //using the node bounds, currX to center the node horizontally and currY for the
node's top edge
    node.setPosition(new DvtDiagramPoint(currX - bounds.x - bounds.w * .5, currY));
    currY += bounds.h + padding;
  }
};

```

A client layout specifying the links connecting the diagram nodes is shown in [Figure 33-17](#).

Figure 33-17 Vertical Node Client Layout with Links



In the layout, the layout context object passed to the layout function contains the information about the number of links using the `getLinkCount()` API, and provides access to each link object using the `getLinkByIndex()` API. After looping through the links to access each link, a call to `setPosition()` API is used to locate the link on the node. The position is passed in the form of a `DvtDiagramPoint()` object which defines the simple x and y coordinates.

The JavaScript code for a vertical client layout with links is here:

```

var DemoVerticalLayoutWithLinks = {
};

/**
 * Main function that does the layout (Layout entry point)
 * @param {DvtDiagramLayoutContext} layoutContext Object that defines a context for
layout call
 */
DemoVerticalLayoutWithLinks.layout = function (layoutContext) {
  //reading optional attributes that could be defined on the page
  var layoutAttrs = layoutContext.getLayoutAttributes();
  var padding = (layoutAttrs && layoutAttrs["nodePadding"]) ?
parseInt(layoutAttrs["nodePadding"]) : 10;

  // layout/position nodes
  var currX = 0;
  var currY = 0;
  var nodeCount = layoutContext.getNodeCount();
  for (var ni = 0; ni < nodeCount; ni++) {
    var node = layoutContext.getNodeByIndex(ni);
    var bounds = node.getContentBounds();
    DemoVerticalLayoutWithLinks.positionNode(node, currX, currY);
    currY += bounds.h + padding;
  }
}

```

```

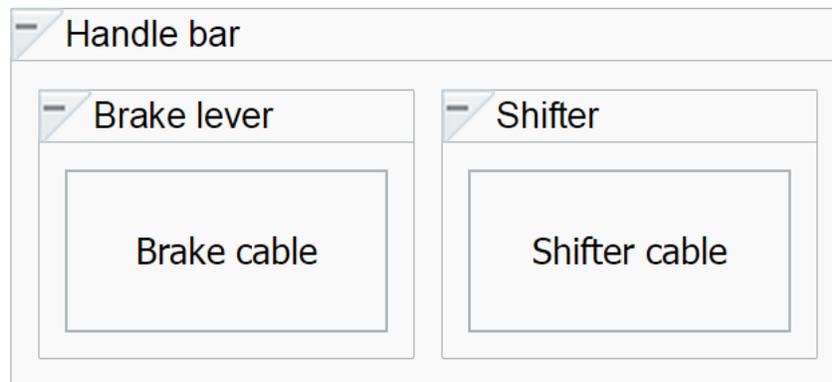
//position links
for (var li = 0;li < layoutContext.getLinkCount();li++) {
  var link = layoutContext.getLinkByIndex(li);
  DemoVerticalLayoutWithLinks.positionLink(layoutContext, link);
}
};

/**
 * Helper function that positions the node using x,y coordinates
 * @param {DvtDiagramLayoutContextNode} node Node to position
 * @param {number} x Horizontal position for the node center
 * @param {number} y Vertical position for the node top edge
 */
DemoVerticalLayoutWithLinks.positionNode = function (node, x, y) {
  var bounds = node.getContentBounds();
  node.setPosition(new DvtDiagramPoint(x - bounds.x - bounds.w * .5, y));
};

```

Figure 33-18 shows a client layout of container nodes with parent and child nodes.

Figure 33-18 Client Layout with Container Nodes



The JavaScript code for a client layout with container nodes is here:

```

var DemoContainersLayout = {};

/**
 * Main function that does the layout (Layout entry point)
 * @param {DvtDiagramLayoutContext} layoutContext Object that defines a context for
layout call
 */
DemoContainersLayout.layout = function(layoutContext) {
  var nodeCount = layoutContext.getNodeCount();
  var currX = 0;
  // iterate through the nodes in the layer and position them horizontally
  for (var ni = 0; ni < nodeCount; ni++) {
    var node = layoutContext.getNodeByIndex(ni);
    node.setPosition(new DvtDiagramPoint(currX, 0));
    currX += node.getBounds().w + 10;
  }
};

```

 **Note:**

- The `setContainerPadding` method sets the container padding. The top, right, bottom, left can be specified as a number or `auto`. You can specify `auto` as the value for the `setContainerPadding` method to prevent the facet information from being truncated.
- The `LayoutBehaviour` attribute indicates whether a specified layout should be used globally to layout the entire diagram at once. The attribute has two possible values, `container` and `global`. With `container`, the nodes and links that belong to the currently processed container will get added to the layout. With `global`, the nodes and links will get added to the specified layout on the diagram. All the other layouts specified on container nodes will be ignored.

You can view and download additional layout examples on the ADF Faces Components Demo application. See [Downloading and Installing the ADF Faces Components Demo Application](#).

34

Using Tag Cloud Components

This chapter describes how to use the ADF Data Visualization Tag Cloud component to display data in tag clouds using simple UI-first development. The chapter defines the data requirements, tag structure, and options for customizing the look and behavior of these components.

If your application uses the Fusion technology stack, you can use data controls to create tag clouds. For more information, see "Creating Databound Tag Clouds" in *Developing Fusion Web Applications with Oracle Application Development Framework*

- [About the Tag Cloud Component](#)
- [Using the Tag Cloud Component](#)

About the Tag Cloud Component

An ADF DVT Tag Cloud is a visual representation for text data, typically used to visualize free form text or tag metadata for a website. Tag clouds are useful for displaying non-prioritized data collections, such as a list of countries or states, or mutually exclusive data in a collection, such as music genres.

The importance or frequency of each tag is shown with font size or color. This visualization is useful for quickly identifying the most prominent terms in a set.

Tag Cloud Use Cases and Examples

Tag Clouds are made up of an array of tags. There are a number of use cases.

Tag clouds are most useful to quickly gauge the relative percentage power of an item in a population. For example, [Figure 34-1](#) shows the results of a survey wherein respondents disclosed which social media networks they use.

Figure 34-1 Simple Tag Cloud with Cloud Layout



Tag Clouds may also be displayed in a rectangular layout. This is most useful for formal data, website tag searches, or if the cloud needs to be contained within a fixed container. [Figure 34-2](#) shows the latest census data for population of the states in the US. The tags are configured to show the population in the tooltip for the tag.

Figure 34-2 Tag Cloud with Rectangular Layout



Tags may be divided into groups, and these groups may be represented in a legend. This is useful for quickly dividing tags by criteria. For example, [Figure 34-3](#) shows US states bifurcated into groups of low or high population. The condition provided classifies states with less than 5 million people as low population and with more than 5 million people as high population. Tag clouds accept both match and exception rules to allow for complex grouping.

Figure 34-3 Tag Cloud with Colored Groups and Legend



End User and Presentation Features of Tag Clouds

Tag Clouds have a number of presentation features that make it a versatile and useful tool for tag representation.

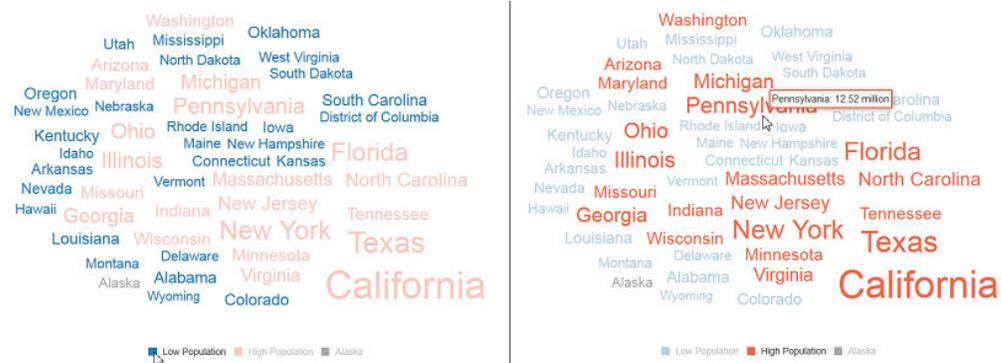
Tag Clouds can be configured to hide and show series using legend items. By clicking on a legend item, all tags mapped to that legend item will be hidden. This is indicated by a hollow square in the legend. [Figure 34-4](#) shows the high and low population tag cloud with the low population states hidden.

Figure 34-4 Tag Cloud Hide and Show Series



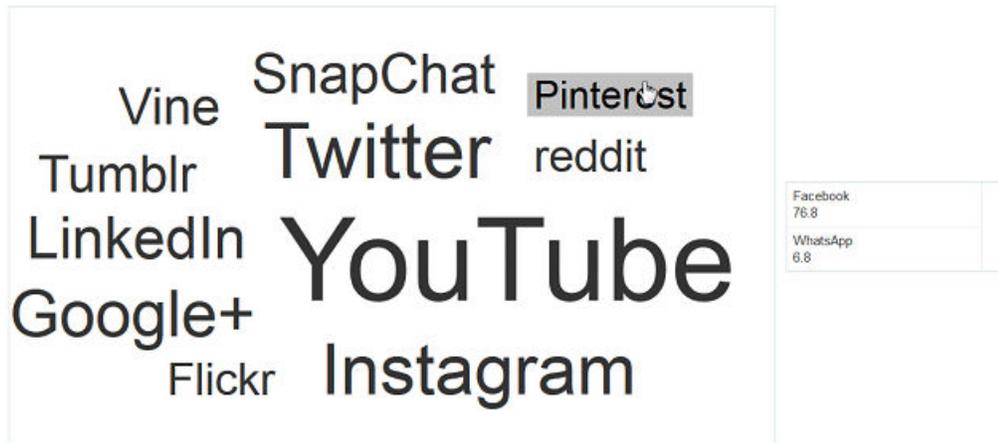
Tag Clouds can be configured to highlight groups by hovering over the tags or the corresponding legend items. [Figure 34-5](#) shows the high and low population tag cloud. In the first case, hovering over the low population legend item dims all other groups. In the second case, hovering over a high population tag dims all other groups.

Figure 34-5 Tag Cloud Highlighting



Tag Cloud items support drag and drop using the `af:dragSource` and `af:dropTarget` attributes. Items may be dragged across any items that support those two attributes. For example, [Figure 34-6](#) shows the social media usage tag cloud, with two items dragged out of the cloud and dropped into a table component.

Figure 34-6 Tag Cloud Drag and Drop



Drag Event triggered on following Tag Cloud items: WhatsApp

Tag Clouds support context menus. Figure 34-7 shows the social media tag cloud configured with a custom context menu to display additional options for each tag.

Figure 34-7 Tag Cloud Context Menu



Tag clouds also support single action behavior, such as linking, via the `action` and `actionListener` attributes, or individual tag links via the `destination` attribute.

Additional Functionality for Tag Cloud Components

You may find it helpful to understand other ADF Faces features before you implement your tag cloud. Additionally, once you have added a tag cloud to your page, you may find that you need to add functionality such as validation and accessibility.

Following are links to other functionality that tag cloud components can use:

- Client-side framework: DVT components are rendered on the client when the browser supports it. You can respond to events on the client using the `af:clientListener` tag. For more information, see [Listening for Client Events](#).
- Partial page rendering: You may want a tag cloud to refresh to show new data based on an action taken on another component on the page. For more information, see [Rerendering Partial Page Content](#).
- Personalization: When enabled, users can change the way the tag cloud displays at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For information, see [Allowing User Customization on JSF Pages](#).
- Accessibility: You can make your tag cloud components accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- Touch devices: When you know that your ADF Faces application will be run on touch devices, the best practice is to create pages specific for that device. For additional information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- Skins and styles: You can customize the appearance of tag cloud components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see [Customizing the Appearance Using Styles and Skins](#).
- Automatic data binding: If your application uses the Fusion technology stack, then you can create automatically bound tag clouds based on how your ADF Business Components are configured. For more information, see "Creating Databound Tag Cloud Components" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

 **Note:**

If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see the "Designing a Page Using Placeholder Data Controls" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Additionally, data visualization components share much of the same functionality, such as how data is delivered, automatic partial page rendering (PPR), and how data can be displayed and edited. For more information, see [Common Functionality in Data Visualization Components](#).

Using the Tag Cloud Component

To use the ADF DVT Tag Cloud component, add the tag cloud to a page using the Component Palette window. Then define the data for the tag cloud and complete the additional configuration in JDeveloper using the tag attributes in the Properties window.

Tag Cloud Data Requirements

Tag clouds require a collection of string items which are represented as the tags or words in the component.

The `label` attribute accepts a string item or a collection of strings to display as tags.

Optionally, you can also provide a numeric value for the tags to the `value` attribute, which will be used to modify the relative size of each tag. You can also provide grouping information using the `dvt:attributeGroups` child tag, wherein tags in the same group will be rendered in the same color.

How to Add a Tag Cloud to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a tag cloud to a JSF page. When you drag and drop a tag cloud component onto the page, the tag cloud is added to your page, and you can use the Properties window to specify data values and configure additional display attributes.

In the Properties window you can click the icon that appears when you hover over the property field to display a property description or edit options. The figure shows the dropdown menu for a tag cloud `value` attribute.



Note:

If your application uses the Fusion technology stack, then you can use data controls to create a Tag Cloud and the binding will be done for you. For more information, see the "Creating Databound Tag Clouds" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of how tag cloud attributes and tag cloud child tags can affect functionality. For more information, see [Configuring Tag Clouds](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Tag Cloud Components](#)

You must complete the following tasks:

1. Create an application workspace as described in [Creating an Application Workspace](#).
2. Create a view page as described in [Creating a View Page](#).

To add a tag cloud to a page:

1. In the ADF Data Visualization page of the Components window, drag and drop a **Tag Cloud** from the Common panel to the page.
2. In the Properties window, view the attributes for the Tag Cloud. Use the **Help** button or press F1 to display the complete tag documentation for the `tagCloud` component.

3. In the Structure window, expand the **dvt:tagCloud** element to find the **dvt:tagCloudItem** element. View and modify the attributes for the **dvt:tagCloudItem** in the Properties window.

What Happens When You Add a Tag Cloud to a Page

JDeveloper generates only a minimal set of tags when you drag and drop a tag cloud from the Components window onto a JSF page.

The generated code is:

```
<dvt:tagCloud id="tc1"/>
```

After inserting a tag cloud component into the page, you can use the visual editor or Properties window to add data or customize tag cloud features. For information about setting component attributes, see [How to Set Component Attributes](#).

If you choose to bind the data to a data control when creating the tag cloud, JDeveloper generates code based on the data model. For more information, see the "Creating Databound Tag Clouds" section in *Developing Fusion Web Applications with Oracle Application Development Framework*

Configuring Tag Clouds

The Tag Cloud component has configurable attributes and child components that you can add or modify to customize the display or behavior of the tag cloud.

The prefix `dvt:` occurs at the beginning of the name of each tag cloud component, indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library. You can configure Tag Cloud child components, attributes, and supported facets in the following areas:

- **Tag Cloud (`dvt:tagCloud`):** Wraps the tag cloud child tags. Configure the following attributes to control the tag cloud display:
 - **hideAndShowBehavior:** Specifies the hide/show behavior when clicking on legend items. This attribute is only applicable when attribute groups are used to color the tag cloud items.
 - **hoverBehavior:** Specifies the behavior when hovering over word cloud or legend items. This attribute is only applicable when attribute groups are used to color the tag cloud items.
 - **layout:** Specifies the built-in layout to use for tag display. The attribute may be set to `rectangular`, wherein items are displayed in a rectangular shape, or `cloud`, wherein items are displayed with weighted words in the center and lighter words towards the edges.
 - **legendSource:** Specifies the id of the `dvt:attributeGroups` used for display in the legend. This `dvt:attributeGroups` element must be a child of a `dvt:tagCloudItem` within this component.
 - **selectionMode:** Specifies whether a tag cloud item can be selected or not. Allows for single or multiple selection.

- Tag Cloud Item (`dvt:tagCloudItem`): Use to define the properties for the displayed data in the tag cloud. Configure the following attributes to control the display of the tag cloud item display:

Table 34-1 Tag Cloud Child Component Attributes

Child Tag Attributes	Description
<code>action</code>	Specifies a reference to an action method sent by the component, or the static outcome of an action.
<code>actionListener</code>	Specifies a method reference for an action listener.
<code>destination</code>	Specifies the URL this component references.
<code>shortDesc</code>	Specifies the custom tooltip text for the tag cloud item.
<code>inlineStyle</code>	Specifies the CSS style for the tag item. It accepts font-related CSS attributes like <code>font-name</code> , <code>font-family</code> , <code>font-weight</code> , <code>font-style</code> , and <code>color</code> . Font-size will be ignored as font-size is determined by the layout.
<code>label</code>	Specifies the text string for this tag cloud item.
<code>value</code>	Specifies the value of this tag cloud item. The value determines the relative size of the node within the tag cloud.

- Tag Cloud Facets: Tag clouds do not support facets.

Part VI

Completing Your View

Part VI explains how to add application-wide functionality to your ADF Faces application, including using skins, localization, accessibility, user customization, drag and drop, print and email output modes, and active data.

Specifically, it contains the following chapters:

- [Customizing the Appearance Using Styles and Skins](#)
- [Internationalizing and Localizing Pages](#)
- [Developing Accessible ADF Faces Pages](#)
- [Allowing User Customization on JSF Pages](#)
- [Adding Drag and Drop Functionality](#)
- [Using Different Output Modes](#)
- [Using the Active Data Service with an Asynchronous Backend](#)

Customizing the Appearance Using Styles and Skins

This chapter describes how to customize the appearance of an ADF application by changing component style attributes or by using ADF skins. Information about the tools that you can use to create ADF skins is also provided in addition to details about how to enable end users to change an application's ADF skin at runtime.

This chapter includes the following sections:

- [About Customizing the Appearance Using Styles and Skins](#)
- [Changing the Style Properties of a Component](#)
- [Enabling End Users to Change an Application's ADF Skin](#)
- [Using Scalar Vector Graphics Image Files](#)

About Customizing the Appearance Using Styles and Skins

ADF skin is a type of CSS (Cascading Style Sheet) file that lets you customize the appearance of the ADF application user interface in ways that are not feasible to define using CSS style properties alone. You can also override the ADF skin properties of the user interface using the style-related properties exposed by individual ADF Faces components.

You can customize the appearance of ADF Faces and ADF Data Visualization components using an ADF skin that you apply to the application or by applying CSS style properties directly to an ADF Faces or ADF Data Visualization component if the component exposes a style-related property (`styleClass` or `inlineStyle`). Choosing the latter option means that you override style properties defined in your application's ADF skin for the component. You might do this when you want to change the style for an instance of a component on a page rather than for all components throughout the application or you want to programmatically set styles conditionally. For example, you want to display text in red only under certain conditions. For more information, see [Changing the Style Properties of a Component](#).

An **ADF skin** is a type of CSS file where you define CSS style properties based on the Cascading Style Sheet (CSS) specification for the component for which you want to customize the appearance. The ADF skin defines rules that determine how to apply CSS style properties to specific components or groups of components. The end user's browser interprets these rules and overrides the browser's default settings. [Figure 35-1](#) and [Figure 35-2](#) demonstrate the result that applying ADF skins can have on the appearance of the ADF Faces and ADF Data Visualization components that appear in your application. [Figure 35-1](#) shows a page from the File Explorer application with the `alta` ADF skin applied.

Figure 35-1 Index Page Using the Alta Skin

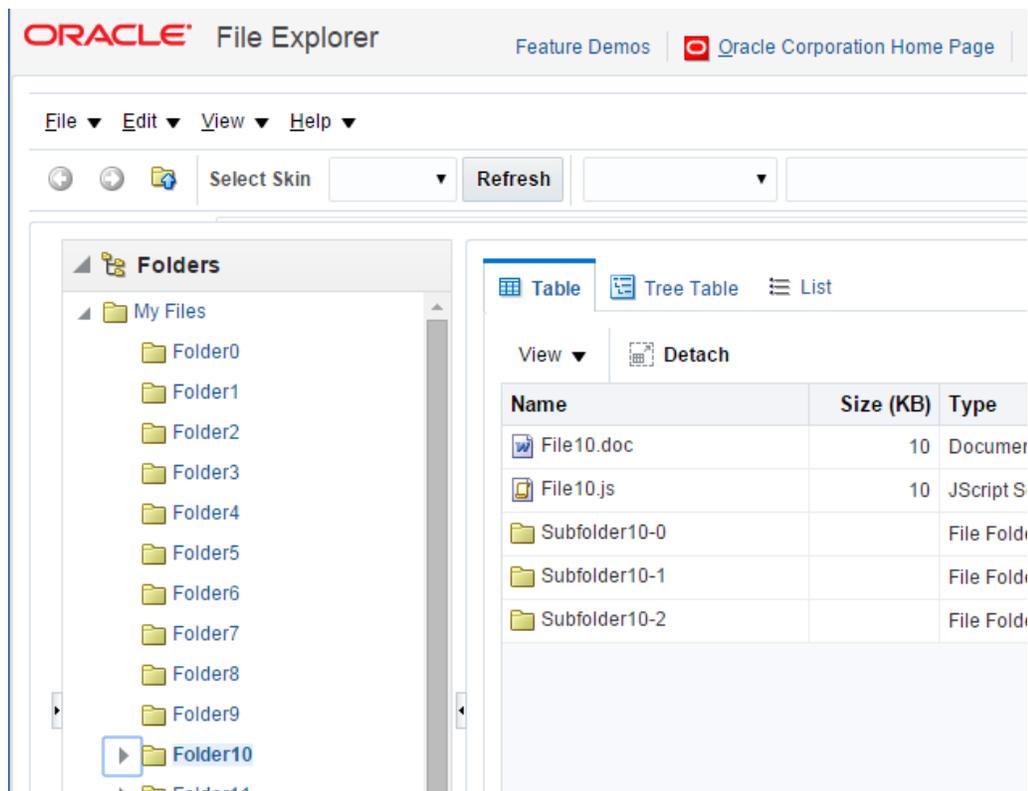
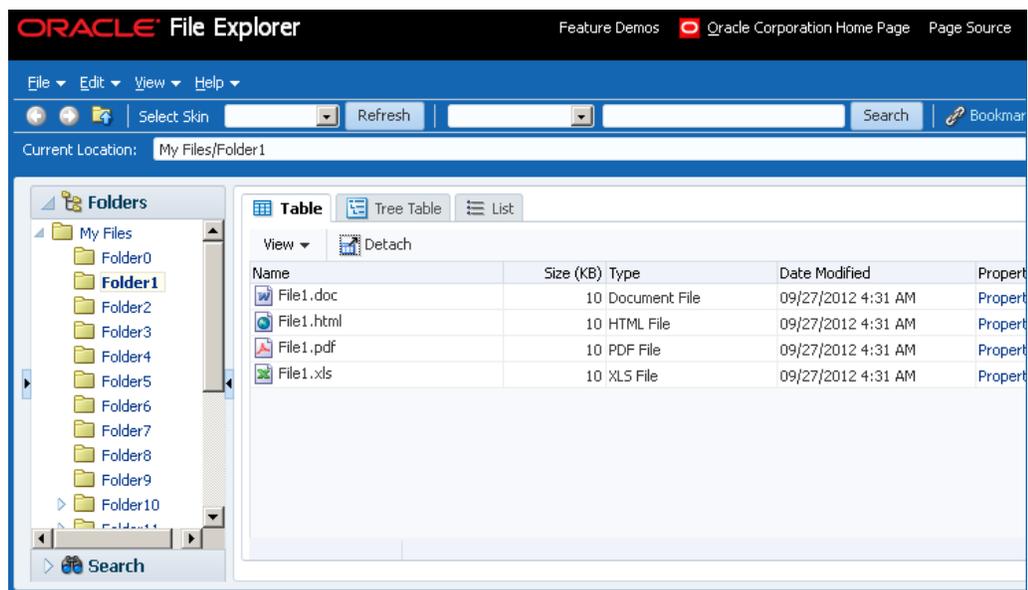


Figure 35-2 shows the same page from the same application with the skyros ADF skin applied.

Figure 35-2 Index Page Using the Skyros Skin



A new web application that you create in this release uses the `alta` skin by default. Migrated web applications continue to use their existing ADF skin. To get the full benefit of the Oracle Alta UI system, Oracle recommends that you go beyond simply using the `alta` skin and design your application around the Oracle Alta UI Design Principles. Designing your application using these principles enables you to make use of the layouts, responsive designs and components the Oracle Alta UI system incorporates to present content to your end users in a clean and uncluttered way. For more information about the Oracle Alta UI system and the Oracle Alta UI Design Principles, see <http://www.oracle.com/webfolder/ux/middleware/alta/index.html> and for information about Oracle Alta UI Patterns, see <http://www.oracle.com/webfolder/ux/middleware/alta/patterns/index.html>.

The File Explorer application provides several ADF skins to customize the appearance of the application. You can view the source files for these ADF skins and the File Explorer application. For more information, see [About the ADF Faces Components Demo Application](#).

It is beyond the scope of this guide to explain the concept of CSS. For extensive information on style sheets, including the official specification, visit the World Wide Web Consortium's website at:

<http://www.w3.org/>

It is also beyond the scope of this guide to describe how to create, modify, or apply ADF skins to your application. Oracle ADF provides editors in JDeveloper and the Theme Editor to assist you with these tasks. See *Working with ADF Skins in JDeveloper* in *Developing ADF Skins*.

If you create multiple ADF skins, you can configure your application so that end users choose the ADF skin that they want the application to use. For more information, see [Enabling End Users to Change an Application's ADF Skin](#).

Customizing the Appearance Use Cases and Examples

You can customize an ADF skin to serve a number of purposes. For example, you might define properties in an ADF skin so that the application highlights a data row rendered by an ADF Faces `table` component after an end user selects it to provide feedback, as illustrated in [Figure 35-3](#).

Figure 35-3 ADF Skin Properties in an ADF Table Component

Number	Name	Size of the file in Kilo Bytes	Number
0	📁 .	0 B	0
1	📁 ..	0 B	1
2	📄 admin.jar	1 KB	2
3	📁 applib	0 B	3
4	📁 applications	0 B	4
5	📁 config	0 B	5
6	📁 connectors	0 B	6
7	📁 database	0 B	7
8	📁 default-we...	0 B	8
9	📄 iiop.jar	1,290 KB	9

Use ADF skin properties to define behavior and appearance that you cannot specify using only CSS or that is dependent on component properties and, as a result, is not feasible to define using CSS. For example, ADF Faces supports animation in browsers where CSS 3 animations are not available. If you want to configure the duration of an animation, use an ADF skin property to do so. [Example 35-1](#) shows how an ADF skin property defines the duration that an ADF Faces `carousel` component displays the spin animation to be 500 milliseconds long.

Example 35-1 Using an ADF Skin Property to Specify Length of Spin Animation

```
af|carousel {  
  -tr-spin-animation-duration: 500;  
}
```

Additional Functionality for Customizing the Appearance

You may find it helpful to understand other ADF Faces features and non-ADF Faces features before you decide to customize the appearance of your application. The following links provide more information that may be useful to know:

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want the text displayed in a component to contain parameters that will resolve at runtime. For more information, see [How to Use the EL Format Tags](#).
- **Internationalization and localization:** The ADF skin that you create to apply to your application can be customized as part of a process to internationalize and localize ADF Faces pages. For more information about this process, see [Internationalizing and Localizing Pages](#).
- **Accessibility:** The ADF skin that you create to apply to your application can be customized as part of a process to make your ADF Faces pages accessible. For more information about this process, see [Developing Accessible ADF Faces Pages](#).
- **Touch Devices:** ADF Faces components may behave and display differently on touch devices. For more information, see [Creating Web Applications for Touch Devices Using ADF Faces](#).
- **Drag and Drop:** You can configure your components so that the user can drag and drop them to another area on the page. For more information, see [Adding Drag and Drop Functionality](#).

Changing the Style Properties of a Component

ADF Faces components expose style-related properties that you can customize to alter the ADF skin properties that determine the look and feel of the ADF application user interface.

You can adjust the look and feel of any component at design time by changing the component's style-related properties, `inlineStyle` and `styleClass`, both of which render on the root DOM element. Any style-related property (`inlineStyle` or `styleClass`) you specify at design time overrides the comparable style specified in the application's ADF skin for that particular instance of the component. Any value you specify for a component's `inlineStyle` property overrides a value set for the `styleClass` attribute.

The `inlineStyle` attribute is a semicolon-delimited string of CSS styles that can set individual attributes, for example, `background-color:red; color:blue; font-`

`style:italic; padding:3px`. The `styleClass` attribute is a CSS style class selector used to group a set of inline styles. The style classes can be defined using an ADF public style class, for example, `.AFInstructionText`, sets all properties for the text displayed in an `af:outputText` component.

 **Note:**

Do not use styles to achieve stretching of components. Using styles to achieve stretching is not declarative and, in many cases, will result in inconsistent behavior across different web browsers. Instead, you can use the geometry management provided by the ADF Faces framework to achieve component stretching. For more information about layouts and stretching, see [Geometry Management and Component Stretching](#).

How to Set an Inline Style

Set an inline style for a component by defining the `inlineStyle` attribute. You can use inline style to specify the style of a component for that instance of the component. For more information, see [Arranging Contents to Stretch Across a Page](#).

Before you begin:

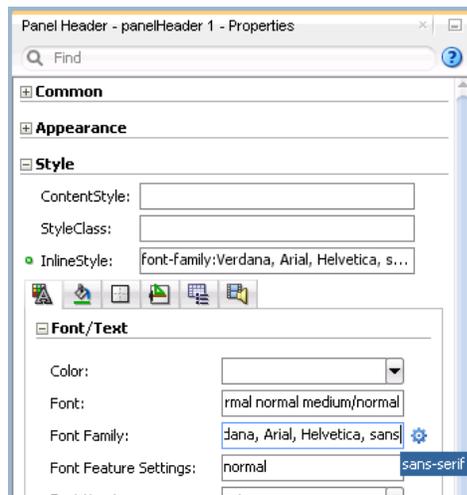
It may be helpful to have an understanding of how the `inlineStyle` attribute relates to other attributes. For more information, see [Changing the Style Properties of a Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Customizing the Appearance](#).

To set an inline style:

1. In the JSF page, select the component for which you want to set an inline style.
2. In the Properties window, expand the **Style** section and enter the inline style you want to use in the **InlineStyle** field.

Alternatively, you can select the style features that you want from dropdown lists, as shown in [Figure 35-4](#).

Figure 35-4 Setting an inlineStyle Attribute

JDeveloper adds the corresponding code for the component to the JSF page. [Example 35-2](#) shows the source for an `af:outputText` component with an `inlineStyle` attribute.

3. You can use an EL expression for the `inlineStyle` attribute itself to conditionally set inline style attributes. For example, if you want the date to be displayed in red when an action has not yet been completed, you could use the code similar to that in [Example 35-3](#).
4. The ADF Faces component may have other style attributes not available for styling that do not register on the root DOM element. For example, for the `af:inputText` component, set the text of the element using the `contentStyle` property, as shown in [Example 35-4](#).

Example 35-2 InlineStyle in the Page Source

```
<af:outputText value="outputText1" id="ot1"
  inlineStyle="color:Red; text-decoration:overline;"/>
```

Example 35-3 EL Expression Used to Set an inlineStyle Attribute

```
<af:outputText value="#{row.assignedDate eq
  null?res['srsearch.unassignedMessage']:row.assignedDate}"
  inlineStyle="#{row.assignedDate eq null?'color:rgb(255,0,0);':''}"
  id="ot3"/>
```

Example 35-4 Using the contentStyle Property

```
<af:inputText value="outputText1"
  contentStyle="color:Red;" id="it1"/>
```

How to Set a Style Class

You can define the style for a component using a style class. You create a style class to group a set of inline styles. Use the `styleClass` attribute to reference the style class.

Before you begin:

It may be helpful to have an understanding of how the `styleClass` attribute relates to other attributes. For more information, see [Changing the Style Properties of a Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Customizing the Appearance](#).

To set a style using a style class:

1. In the JSF page, select the component for which you want to define a style.
2. In the Properties window, expand the **Style** section and enter the name of the style class that you want the component to use in the **StyleClass** field.

[Example 35-5](#) shows an example of a style class being used in the page source.

3. You can also use EL expressions for the `styleClass` attribute to conditionally set style attributes. For example, if you want the date to be displayed in red when an action has not yet been completed, you could use code similar to that in [Example 35-3](#).

Example 35-5 Page Source for Using a Style Class

```
<af:outputText value="Text with a style class"
  styleClass="overdue" id="ot4"/>
```

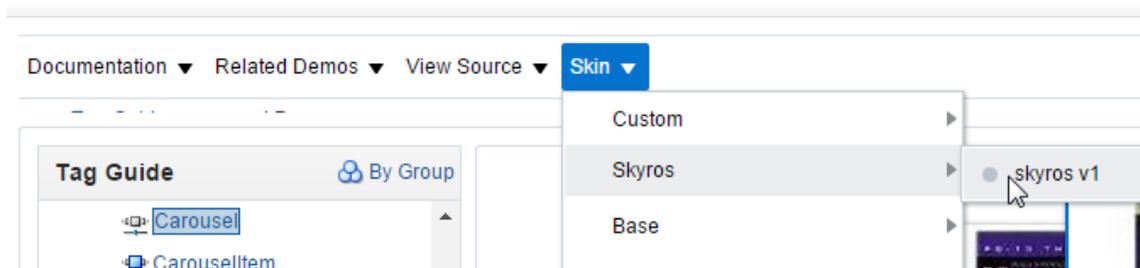
Enabling End Users to Change an Application's ADF Skin

The ADF skins named in the ADF application `trinidad-config.xml` file can be exposed to end users so that they may change the ADF application look and feel to suit their locale.

You can configure your application to enable end users select an alternative ADF skin. You might configure this functionality when you want end users to render the application's page using an ADF skin that is more suitable for their needs. For example, you want your application to use an ADF skin with features specific to a Japanese locale when a user's browser is Japanese. An alternative example is where you want your application to use an ADF skin that is configured to make your application's pages more accessible for users with disabilities.

[Figure 35-5](#) shows how you might implement this functionality by configuring a component (for example, `af:commandMenuItem`) to allow end users to change the ADF skin the application uses at runtime. Configure the component on the JSF page to set a scope value that can later be evaluated by the `skin-family` property in the `trinidad-config` file.

Figure 35-5 Changing an Application's ADF Skin



How to Enable End Users Change an Application's ADF Skin

You enable end users change an application's ADF skin by exposing a component that allows them to update the value of the `skin-family` property in the `trinidad-config` file.

Before you begin:

It may be helpful to have an understanding of how the changes that you make can affect functionality. For more information, see [Enabling End Users to Change an Application's ADF Skin](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Customizing the Appearance](#).

To enable end users change an application's ADF skin:

1. Open the main JSF page where you want to configure the component(s) that you use to set the skin family.
2. Configure a number of components (for example, `af:commandMenuItem` components) that allow end users to choose one of a number of available ADF skins at runtime, as shown in [Figure 35-5](#).

[Example 35-6](#) shows how you configure `af:commandMenuItem` components that allow end users to choose available ADF skins at runtime, as shown in [Figure 35-5](#). Each `af:commandMenuItem` component specifies a value for the `actionListener` attribute. This attribute passes an `actionEvent` to a method (`skinMenuAction`) on a managed bean named `skins` if an end user clicks the menu item.

3. Write a method (for example, `skinMenuAction`) on a managed bean named `skins` to store the value of the ADF skin selected by the end user. [Example 35-7](#) shows a method that takes the value the end user selected and uses it to set the value of `skinFamily` in a managed bean. The method in [Example 35-7](#) also invokes a method to reload the page after it sets the value of `skinFamily`.
4. In the Applications window, double-click the `trinidad-config.xml` file.
5. In the `trinidad-config.xml` file, write an EL expression to dynamically evaluate the skin family:

```
<skin-family>#{skins.skinFamily}</skin-family>
```

Example 35-6 Using a Component to Set the Skin Family

```
<af:menu text="Change Skin" id="skins" binding="#{backing_ChangeSkin.skins}">
  <af:commandMenuItem id="skin1" text="skyros" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='skyros'}"/>
  <af:commandMenuItem id="skin3" text="fusion" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='fusion'}"/>
  <af:commandMenuItem id="skin4" text="fusion-projector" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='fusion-projector'}"/>
  <af:commandMenuItem id="skin5" text="simple" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='simple'}"/>
</af:menu>
```

```
<af:commandMenuItem id="skin6" text="skin1" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='skin1'}"/>
</af:menu>
```

Example 35-7 Managed Bean Method to Change ADF Skin

```
public void skinMenuAction(ActionEvent ae)
{
    RichCommandMenuItem menuItem = (RichCommandMenuItem)ae.getComponent();

    // Invoke another managed bean method to set the value of skinFamily
    setSkinFamily(menuItem.getText());

    // Invoke a method to reload the page. The application reloads the page
    // using the newly selected ADF skin.
    reloadThePage();
}
```

What Happens at Runtime: How End Users Change an Application's ADF Skin

At runtime, the end user uses the component that you exposed to select another ADF skin. In [Example 35-6](#), this is one of a number of `af:commandMenuItem` components. This component submits the value that the end user selected to a managed bean that, in turn, sets the value of a managed bean property (`skinFamily`). At runtime, the `<skin-family>` property in the `trinidad-config` file reads the value from the managed bean using an EL expression.

Using Scalar Vector Graphics Image Files

Scalar Vector Graphics (SVG) images files are used to define vector-based graphics for the web in XML format. Use inline SVG images via CSS and Javascript for the ADF Faces components to access various elements available within the SVG DOM.

ADF Faces supports the use of SVG image files to render icons. ADF Faces components that reference this type of image render an HTML `` tag in the generated page at runtime.

As an alternative to HTML `` tag, some ADF Faces components expose an additional `iconDelivery` attribute that you can set to render a page that uses an HTML `<svg>` tag rather than an HTML `` tag. Rendering the HTML `<svg>` tag gives SVG authors more control over the rendered SVG image file, such as, for example, color changes based on an alias in a skin.

You can set the `iconDelivery` attribute to the following values:

- `auto`: ADF Faces framework handles the SVG image rendering internally. When set to `auto`, user can override the framework behavior by setting the `-tr-icon-delivery` skin property to `reference/inline/fetchAndInline` values.
- `reference`: When set to `reference`, HTML `` tag is rendered. By default, the `iconDelivery` attribute is set to `reference`.
- `inline` and `fetchAndInline`: When set to `inline` or `fetchAndInline`, HTML `<SVG>` tag is rendered.

Set to `fetchAndInline` if you want to allow browsers to cache the SVG file while still being able to inline the SVG content in the HTML page. Use of the `inline` or `fetchAndInline` values for the `iconDelivery` attribute only works for SVG images.

The following components support SVG `inline` and `fetchAndInline` values for the `iconDelivery` attribute:

- RichActiveCommandToolBarButton
- RichActiveImage
- RichCommandButton
- RichCommandImageLink
- RichCommandMenuItem
- RichCommandNavigationItem
- RichCommandToolBarButton
- RichGoButton
- RichGoImageLink
- RichGoMenuItem
- RichImage
- RichMenu
- RichPanelBox
- RichShowDetailItem
- RichIcon

**Note:**

The RichIcon component supports only the `inline` value for the `iconDelivery` attribute and does not support `fetchAndInline` value.

An example of a component that exposes the `iconDelivery` attribute is `showDetailItem`. See *Tag Reference for Oracle ADF Faces*.

What You May Need to Know About Inline SVG Support in ADF Faces

ADF supports styling inline Scalar Vector Graphics (SVG) by adding a style class in skin file and referencing it in the SVG content. This feature is required only when it is desirable to style SVG icons which will be used across the pages and components of the application in a specific way for one or more instances.

As ADF now supports inline SVG icon, you can render SVG icons directly into the ADF page's DOM so that the SVG icons can use the styles from the ADF skin. For example, if you want to style an SVG icon based on the style class from ADF skin, you can add `TestSVGIcon` style in the skin file and reference the same in the SVG content.

See [Working with Style Classes](#) in the *Fusion Middleware Skin Editor User's Guide for Oracle Application Development Framework*.

In ADF Faces, if you want to style a particular component or a component instance added to a page in a different way from what is present in the ADF skin, you can do so by using the `styleClass` attribute. By default, ADF Faces framework enables CSS content compression for better performance and as a result, the CSS selectors are compressed both in the generated CSS and in the DOM. In inline SVG, since the content is fetched and inlined from an external file, the class reference will not be compressed, whereas the `styleClass` attribute value in the DOM is compressed by the framework. This results in a mismatch between what is being referenced and what is present in the generated CSS. Therefore, just adding a style class to the component instance in inline SVG is not sufficient.

If you want to style an SVG icon based on the component container `styleClass`, for example, if you have a command button in a page with inline SVG and want to style this button differently by changing the background fill only in some pages, you can use the `styleClass` attribute. When you have the `styleClass` attribute on a button that renders inline SVG, which in turn references a style class `TestSVGIcon`, there will be a mismatch due to compression in DOM. The style class gets rendered as `class=x20f` in the DOM and the inline SVG content is rendered as `class=TestSVGIcon`.

Take for example, the following CSS, where `redGraphics` is added to the ADF button container and `svg-icon01` is the style class used inside SVG DOM:

```
.redGraphics .svg-icon01 {
fill: #ff0000;
}
```

The CSS generated by the framework will have both the normal and short versions of the style as in the code below:

```
.redGraphics .svg-icon01, .x2he .x2hf {
fill: #ff0000;
}
```

However, in the DOM, there is a mismatch as style class added to button root DOM is compressed while style used in SVG is not. As a result the SVG icon's background-fill color is not applied when rendered. The `redGraphics` container style class is compressed during rendering and applied to DOM, while the `svg-icon01` style class in SVG DOM is not compressed as in the following code.

```
<div class="x2he">
  <svg class="svg-icon01"/>
</div>
```

To avoid this mismatch and style the inline SVG icon based on the component container style class, use the `@no-compress` atRule to declare no CSS compression for particular styles defined within `@no-compress`. ADF Faces provides a no-compress server side atRule, `@no-compress`, to render certain user provided style classes without compressing the `styleClass` attributes so that the style classes appear in the DOM without change. See the Apache Trinidad Skinning page on the Apache MyFaces

website for the list of server-side atRules at <https://myfaces.apache.org/trinidad/devguide/skinning.html#server-side-at-rules>.

The `@no-compress` rule is applicable only for non-namespace selectors to indicate that these selectors are not compressed both in the generated CSS and in the DOM. The selectors that are defined in `@no-compress` rule in the CSS file of the application skin ensures that certain style classes are not compressed by ADF Faces framework at runtime. This is also useful if these selectors are referred to elsewhere in the application. Style definitions once defined as `no-compress` will never be compressed at any level regardless of another rule, such as agent, platform and so on and the order in which they are generated.

The following example does not compress the selector in CSS or in the DOM.

```
@no-compress {
  .TestSVGIcon {
    border-color: Fuchsia;
  }
}
```

Since `no-compress` is not supported for namespace selectors, in the following example, the selector will be written to CSS as `.x1m5{color: red;}` and compressed in both the DOM and generated CSS.

```
@no-compress {
  af|button::link {
    color: red;
  }
}
```

If there is a combination of namespace and non-namespace selectors, then only the non-namespace part is written without any compression. The non-namespace selector in the following example is written to CSS as `.x77 .TextHighlight {font-weight: bold;}`.

```
@no-compress {
  af|button::link .TextHighlight {
    font-weight: bold;
  }
}
```

Assume that your page contains a `commandToolBarButton` with a root **styleclass** as `NoCompressStyle` that is applied on the component root. The SVG edit icon is inlined to display on the button. The following example displays the JSF snippet for the button.

```
<af:commandButton text="Edit" id="cb1" styleClass="NoCompressStyle" icon="skins/
edit.svg"
  iconDelivery="inline" iconPosition="leading"/>
```

In the above example, the `edit.svg` file contains the other style class `TestSvgIcon`.

Page without the `no-compress` rule

ADF Faces, by default enables CSS content compression. Therefore, the class names are compressed in the generated HTML DOM and style class name in the SVG file is not compressed. This results in styles not getting applied on the SVG icon. The generated CSS will contain the styles like `.xvf .xbv {}` and the uncompressed version `.NoCompressStyle .TestSvgIcon` as in the following example.

```
.NoCompressStyle {
  border-color: red;
}
.NoCompressStyle .TestSVGIcon {
  height: 16px;
  width: 16px;
  background:
transparent;
}
```

The generated HTML DOM is given below where the class names are marked in bold.

```
<button id="cb1" class="x20n x7j" onclick="return false;">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px" class="xfb TestSVGIcon"
width="100px" height="100px" viewBox="0 0 100 100" enable-background="new 0 0 100
100" xml:space="preserve"><g> <g>
<path
d="M67.041,78.553h-6.49h0c-0.956,0-1.73,0.774-1.73,1.73h-0.007v3.01H18.173V36.16h17.7
23c0.956,0,1.73-0.774,1.73-1.73
V16.707h21.188l0,12.34h0.016c0.047,0.913,0.796,1.641,1.721,1.641h6.49c0.925,0,1.674-0
.728,1.721-1.641h0.009v-0.088
c0,0-0.001,0-0.001c0-0.001,0-0.001,0-0.001,0-0.002l0-16.457h-0.005V8.48c0-0.956-0.774-1.73-
1.73-1.73h-2.45v0H35.895v0h-1.73
L8.216,32.7v2.447v1.013v52.912v2.447c0,0.956,0.774,1.73,1.73,1.73h1.582h53.925h1.582c
0.956,0,1.73-0.774,1.73-1.73v-2.448
h0.005l0-8.789l0-0.001c68.771,79.328,67.997,78.553,67.041,78.553z"></path> </
g> <g>
<path
d="M91.277,39.04L79.656,27.419c-0.676-0.676-1.771-0.676-2.447,0L45.404,59.224l0.069,0
.069l-0.109-0.029l-4.351,16.237
l0.003,0.001c-0.199,0.601-0.066,1.287,0.412,1.765c0.528,0.528,1.309,0.638,1.948,0.341
l0.002,0.006l16.08-4.309l-0.01-0.037
l0.023,0.024l31.806-31.806C91.953,40.811,91.953,39.716,91.277,39.04z
M46.305,72.353l2.584-9.643l7.059,7.059L46.305,72.353z"></path>
</g></g>
</svg>
<span title="" class="xfe">Edit</span>
</button>
```

Page with no-compress rule applied on both the style classes in the skin file

The following example displays the generated CSS for a page where a no-compress rule is applied and the `.NoCompressStyle .TestSvgIcon {}` style is applied on the SVG icon which is inlined. These style class names are not compressed even in compression mode, which is the default mode in production.

```
@no-compress {
  .NoCompressStyle {
    border-color: red;
  }

  .NoCompressStyle .TestSVGIcon {
    height: 16px;
    width: 16px;
    background: transparent;
  }
}
```

The example HTML DOM is given below where the class names are marked in bold.

```

<button id="cb1" class="NoCompressStyle x7j" onclick="return false;">
  <svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px" class="xfb TestSVGIcon"
  width="100px" height="100px" viewBox="0 0 100 100" enable-background="new 0 0 100
100" xml:space="preserve"><g>
  <g>
  <path
d="M67.041,78.553h-6.49h0c-0.956,0-1.73,0.774-1.73,1.73h-0.007v3.01H18.173V36.16h17.7
23c0.956,0,1.73-0.774,1.73-1.73
V16.707h21.188l0,12.34h0.016c0.047,0.913,0.796,1.641,1.721,1.641h6.49c0.925,0,1.674-0
.728,1.721-1.641h0.009v-0.088
c0,0-0.001,0-0.001c0-0.001,0-0.001,0-0.002l0-16.457h-0.005V8.48c0-0.956-0.774-1.73-
1.73-1.73h-2.45v0H35.895v0h-1.73
L8.216,32.7v2.447v1.013v52.912v2.447c0,0.956,0.774,1.73,1.73,1.73h1.582h53.925h1.582c
0.956,0,1.73-0.774,1.73-1.73v-2.448
h0.005l0-8.789l0-0.001C68.771,79.328,67.997,78.553,67.041,78.553z"></path>
  </g>
  <g>
  <path
d="M91.277,39.04L79.656,27.419c-0.676-0.676-1.771-0.676-2.447,0L45.404,59.224l0.069,0
.069l-0.109-0.029l-4.351,16.237
l0.003,0.001c-0.199,0.601-0.066,1.287,0.412,1.765c0.528,0.528,1.309,0.638,1.948,0.341
l0.002,0.006l16.08-4.309l-0.01-0.037
l0.023,0.024l31.806-31.806C91.953,40.811,91.953,39.716,91.277,39.04z
M46.305,72.353l2.584-9.643l7.059,7.059L46.305,72.353z"></path>
  </g></g>
  </svg>
  <span title="" class="xfe">Edit</span>
</button>

```

Internationalizing and Localizing Pages

This chapter describes how to configure an ADF application so that the application can be used in a variety of locales and international user environments. Key features such as JDeveloper's ability to automatically generate resource bundles, how you can manually define resource bundles and locales in addition to configuring ADF Faces localization properties for your application are also described.

This chapter includes the following sections:

- [About Internationalizing and Localizing ADF Faces Pages](#)
- [Using Automatic Resource Bundle Integration in JDeveloper](#)
- [Manually Defining Resource Bundles and Locales](#)
- [Configuring Pages for an End User to Specify Locale at Runtime](#)
- [Configuring Optional ADF Faces Localization Properties](#)

About Internationalizing and Localizing ADF Faces Pages

JDeveloper simplifies the process of providing locale-specific resources, such as the translatable text displayed by the ADF Faces components, as well as text for messages, by grouping the text in a base resource bundle that you may version for any number of locales.

Internationalization is the process of designing and developing products for easy adaptation to specific local languages and cultures. Localization is the process of adapting a product for a specific local language or culture by translating text and adding locale-specific components. A successfully localized application will appear to have been developed within the local culture. JDeveloper supports easy localization of ADF Faces components using the abstract class `java.util.ResourceBundle` to provide locale-specific resources.

When your application will be viewed by users in more than one country, you can configure your JSF page or application to use different locales so that it displays the correct language for the language setting of a user's browser. For example, if you know your page will be viewed in Italy, you can localize your page so that when a user's browser is set to use the Italian language, text strings in the browser page appear in Italian.

ADF Faces components may include text that is part of the component, for example the `af:table` component uses the resource string `af_table.LABEL_FETCHING` for the message text that displays in the browser while the `af:table` component fetches data during the initial load of data or while the user scrolls the table. JDeveloper provides automatic translation of these text resources into 28 languages. These text resources are referenced in a resource bundle. You must enable support for each language that you want your application to support by specifying the `<supported-locale>` element in the `faces-config.xml` file. For example, if you set the browser to use the language in Italy, and you add `<supported-locale>it</supported-locale>` to the `faces-config.xml` file, any text contained within the components automatically displays in Italian. For more information, see [How to Register a Locale for Your Application](#).

For any text you add to a component, for example if you define the label of an `af:button` component by setting the `text` attribute, you must provide a resource bundle that holds the actual text, create a version of the resource bundle for each locale, and add a `<locale-config>` element to define default and support locales in the application's `faces-config.xml` file. You must also add a `<resource-bundle>` element to your application's `faces-config.xml` file in order to make the resource bundles available to all the pages in your application. Once you have configured and registered a resource bundle, the Expression Builder displays the key from the bundle, making it easier to reference the bundle in application pages.

To simplify the process of creating text resources for text you add to ADF Faces components, JDeveloper supports automatic resource bundle synchronization for any translatable string in the visual editor. When you edit components directly in the visual editor or in the Properties window, text resources are automatically created in the base resource bundle. For more information, see [Using Automatic Resource Bundle Integration in JDeveloper](#).

**Note:**

Any text retrieved from the database is not translated. This document explains how to localize static text, not text that is stored in the database.

Internationalizing and Localizing Pages Use Cases and Examples

Assume, for example, that you have a panel box with a title of My Purchase Requests. Rather than set the literal string, `My Purchase Requests`, as the value of the `af:panelBox` component's `text` attribute, you bind the value of the `text` attribute to a key in the `UIResources` resource bundle. The `UIResources` resource bundle is registered in the `faces-config.xml` file for the application, as shown in the following example.

```
<resource-bundle>
  <base-name>resources.UIResources</base-name>
  <var>res</var>
</resource-bundle>
```

The resource bundle is given a variable name (in this case, `res`) that can then be used in EL expressions. On the page, the `text` attribute of the `af:panelBox` component is then bound to the `myDemo.pageTitle` key in that resource bundle, as shown in the following example.

```
<af:panelBox text="#{res['myDemo.pageTitle']}"
  ....
  id="pbl">
```

The `UIResources` resource bundle has an entry in the English language for all static text displayed on each page in the application, as well as for text for messages and global text, such as generic labels. [Example 36-1](#) shows the keys for the `myDemo` page. [Example 36-2](#) shows the resource bundle version for the Italian (Italy) locale, `UIResources_it`. Note that there is no entry for the selection facet's title, yet it was translated from *Select* to *Seleziona* automatically. That is because this text is part of the ADF Faces `table` component's selection facet.

Note that text in the banner image and data retrieved from the database are not translated.

Example 36-1 Resource Bundle Keys for the myDemo Page Displayed in English

```
#myDemo Screen
myDemo.pageTitle=My Purchase Requests
myDemo.menuBar.openLink=Open Requests
myDemo.menuBar.pendingLink=Requests Awaiting customer
```

Example 36-2 Resource Bundle Keys for the myDemo Page Displayed in Italian

```
#myDemo Screen
myDemo.pageTitle=Miei Ticket
myDemo.menuBar.openLink=Ticket Aperti
myDemo.menuBar.pendingLink=Ticket in Attesa del Cliente
```

Additional Functionality for Internationalizing and Localizing Pages

You may find it helpful to understand other ADF Faces features before you internationalize or localize your application. Additionally, once you have internationalized or localized your application, you may find that you need to add functionality such as accessibility or render ADF Faces components from right to left. Following are links to other functionality that you can use.

- **Using parameters in text:** You can use the ADF Faces EL format tags if you want the text displayed in a component to contain parameters that will resolve at runtime. For more information, see [How to Use the EL Format Tags](#).
- **Pseudo-classes:** ADF skins support a number of pseudo-classes that you can use to change how an application renders in a particular locale. For example, the `:rtl` pseudo-class renders ADF Faces components from right to left. This may be useful if your application is localized into languages, such as Arabic and Hebrew, that read from right to left. For more information, see [Customizing the Appearance Using Styles and Skins](#).
- **Accessibility:** You can make the pages in your application accessible. For more information, see [Developing Accessible ADF Faces Pages](#).
- **Global resource strings:** The default resource bundle stores the global resource strings that ADF Faces components support and the resource strings that are specific to individual components. For information about these resource strings, see the *Tag Reference for Oracle ADF Faces Skin Selectors*.
- **Touch Devices:** ADF Faces components may behave and display differently on touch devices. For more information, see [Creating Web Applications for Touch Devices Using ADF Faces](#)
- **Drag and Drop:** You can configure your components so that the user can drag and drop them to another area on the page. For more information, see [Adding Drag and Drop Functionality](#)

Using Automatic Resource Bundle Integration in JDeveloper

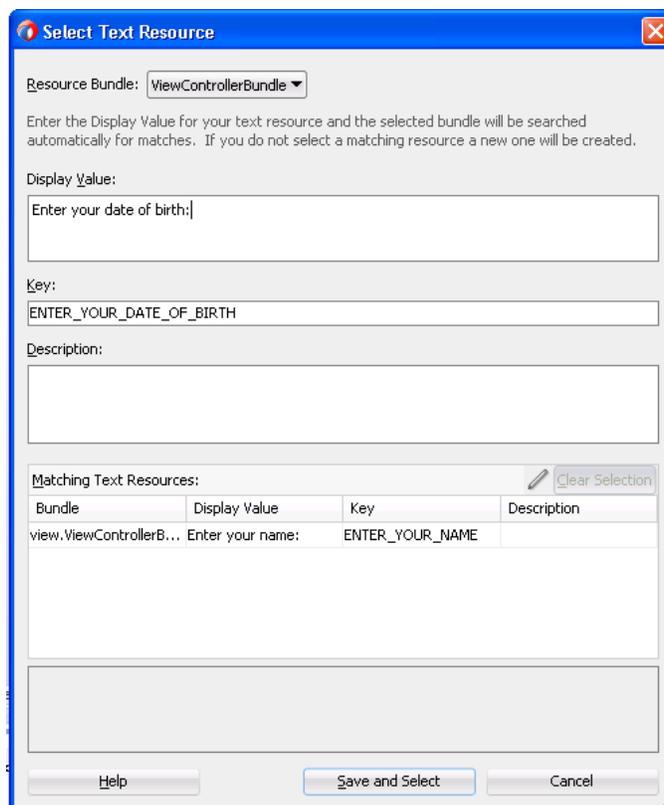
JDeveloper simplifies the process of creating text resources that you add to ADF Faces components, using a text resource editor to synchronize translatable strings with the base resource bundle.

JDeveloper supports the automatic creation of text resources in the default resource bundle when editing ADF Faces components in the visual editor. To treat user-defined strings as static values, clear the **Automatically Synchronize Bundle** checkbox in the Project Properties dialog, as described in [How to Set Resource Bundle Options](#).

Automatic resource bundle integration can be configured to support one resource bundle per page or project.

You can edit translatable text strings using the Select Text Resource dialog shown in [Figure 36-1](#). The dialog can be accessed from the Properties window by clicking the icon that appears when you hover over the property field of a translatable property and select **Select Text Resource** from the context menu. For more information on using this dialog, see [How to Create an Entry in a JDeveloper-Generated Resource Bundle](#).

Figure 36-1 Select Text Resource Dialog



How to Set Resource Bundle Options

You can set resource bundle options for your project in the Project Properties dialog.

Before you begin:

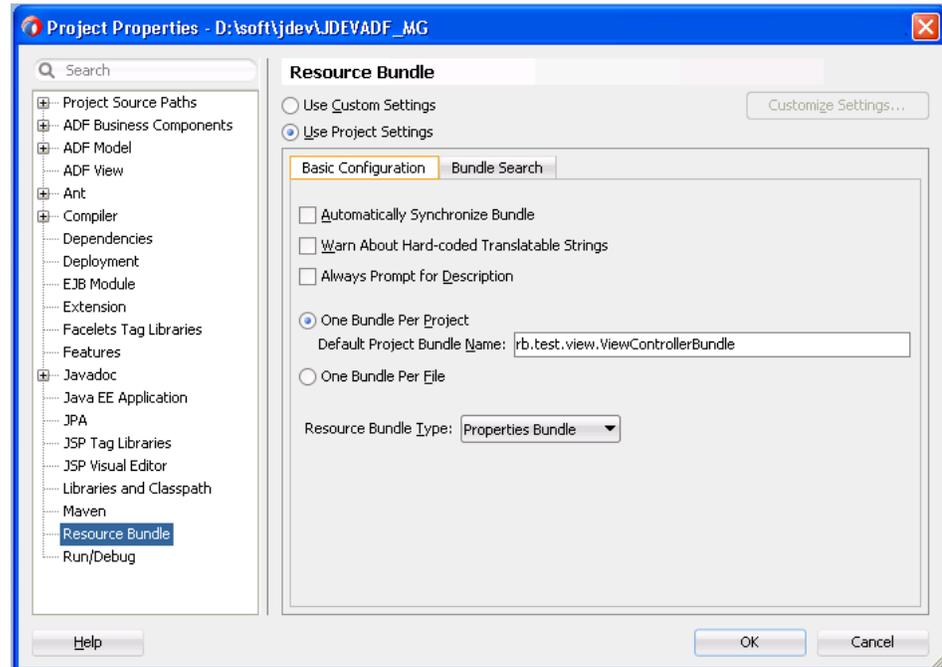
It may help to understand how JDeveloper manages resource bundles. For more information, see [Using Automatic Resource Bundle Integration in JDeveloper](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To set resource bundle options for a project:

1. In the Applications window, double-click the project.
2. In the Project Properties dialog, select **Resource Bundle** to display the resource bundle options, as shown in [Figure 36-2](#).

Figure 36-2 Project Properties Resource Bundle dialog



3. If you want JDeveloper to automatically generate a default resource file, select **Automatically Synchronize Bundle**.
4. Select one of the following resource bundle file options:
 - **One Bundle Per Project** - configured in a file named `<ProjectName>.properties`.
 - **One Bundle Per File** - configured in a file named `<FileName>.properties`.
5. Select the resource bundle type from the dropdown list:
 - **List Resource Bundle**
 - **Properties Bundle**
 - **XML Localization Interchange File Format (XLIFF) Bundle**
6. Click **OK**.

What Happens When You Set Resource Bundle Options

JDeveloper generates one or more resource bundles of a particular type based on the selections that you make in the resource bundle options part of the Project Properties dialog, as illustrated in [Figure 36-2](#). It generates a resource bundle the first time that you invoke the Select Text Resource dialog, as illustrated in [Figure 36-1](#).

Assume, for example, that you select the **One Bundle Per Project** checkbox and the **List Resource Bundle** value from the **Resource Bundle Type** dropdown list. The first

time that you invoke the Select Text Resource dialog, JDeveloper generates one resource bundle for the project. The generated resource bundle is a Java class named after the default project bundle name in the Project Properties dialog (for example, `ViewControllerBundle.java`).

JDeveloper generates a resource bundle as an `.xlf` file if you select the **XML Localization Interchange File Format (XLIFF) Bundle** option and a `.properties` file if you select the **Properties Bundle** option.

By default, JDeveloper creates the generated resource bundle in the view subdirectory of the project's Application Sources directory.

How to Create an Entry in a JDeveloper-Generated Resource Bundle

JDeveloper generates one or more resource bundles based on the values you select in the resource bundle options part of the Project Properties dialog. It generates a resource bundle the first time that you invoke the Select Text Resource dialog from a component property in the Properties window.

JDeveloper writes key-value pairs to the resource bundle based on the values that you enter in the Select Text Resource dialog. It also allows you to select an existing key-value pair from a resource bundle to render a runtime display value for a component.

Before you begin:

It may help to understand how JDeveloper manages resource bundles. For more information, see [Using Automatic Resource Bundle Integration in JDeveloper](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To create an entry in the resource bundle generated by JDeveloper:

1. In the JSF page, select the component for which you want to write a runtime value.
For example, select an `af:inputText` component.
2. In the Properties window, select **Select Text Resource** from the context menu that appears when you click the icon that appears when you hover over a property field to create a new entry in the resource bundle.

The **Select Text Resource** entry in the context menu only appears for properties that support text resources. For example, the **Label** property of an `af:inputText` component.
3. Write the value that you want to appear at runtime in the **Display Value** input field, as illustrated in [Figure 36-1](#).

JDeveloper generates a value in the Key input field.
4. Optionally, write a description in the **Description** input field.

Note:

JDeveloper displays a matching text resource in the Matching Text Resource field if a text resource exists that matches the value you entered in the Display Value input field.

5. Click **Save and Select**.

What Happens When You Create an Entry in a JDeveloper-Generated Resource Bundle

JDeveloper writes the key-value pair that you define in the Select Text Resource dialog to the resource bundle. The options that you select in the resource bundle options part of the Project Properties dialog determine what type of resource bundle JDeveloper writes the key-value pair to. For more information, see [What Happens When You Set Resource Bundle Options](#).

The component property for which you define the resource bundle entry uses an EL expression to retrieve the value from the resource bundle at runtime. For example, an `af:inputText` component's `Label` property may reference an EL expression similar to the following:

```
#{viewControllerBundle.Label1}
```

where `viewControllerBundle` references the resource bundle and `Label1` is the key for the runtime value.

Manually Defining Resource Bundles and Locales

The ADF application supports creating resource bundles as a Java class, property file, or XLIFF file when you need to define text resources outside of the JDeveloper editors.

A resource bundle contains a number of named resources, where the data type of the named resources is `String`. A bundle may have a parent bundle. When a resource is not found in a bundle, the parent bundle is searched for the resource. Resource bundles can be either Java classes, property files, or XLIFF files. The abstract class `java.util.ResourceBundle` has two subclasses:

- `java.util.PropertyResourceBundle`
- `java.util.ListResourceBundle`

A `java.util.PropertyResourceBundle` is stored in a property file, which is a plain-text file containing translatable text. Property files can contain values only for `String` objects. If you need to store other types of objects, you must use a `java.util.ListResourceBundle` class instead.

For more information about using XLIFF, see <http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html>

To add support for an additional locale, replace the values for the keys with localized values and save the property file, appending a language code (mandatory) and an optional country code and variant as identifiers to the name, for example, `UIResources_it.properties`.

The `java.util.ListResourceBundle` class manages resources in a name and value array. Each `java.util.ListResourceBundle` class is contained within a Java class file. You can store any locale-specific object in a `java.util.ListResourceBundle` class. To add support for an additional locale, you create a subclass from the base class, save it to a file with a locale or language extension, translate it, and compile it into a class file.

The `ResourceBundle` class is flexible. If you first put your locale-specific `String` objects in a `java.util.PropertyResourceBundle` file, you can still move them to a `ListResourceBundle` class later. There is no impact on your code, because any call to find your key will look in both the `java.util.ListResourceBundle` class and the `java.util.PropertyResourceBundle` file.

The precedence order is class before properties. So if a key exists for the same language in both a class file and a property file, the value in the class file will be the value presented to you. Additionally, the search algorithm for determining which bundle to load is as follows:

1. (baseclass)+(specific language)+(specific country)+(specific variant)
2. (baseclass)+(specific language)+(specific country)
3. (baseclass)+(specific language)
4. (baseclass)+(default language)+(default country)+(default variant)
5. (baseclass)+(default language)+(default country)
6. (baseclass)+(default language)

For example, if your browser is set to the Italian (Italy) locale and the default locale of the application is US English, the application attempts to find the closest match, looking in the following order:

1. `it_IT`
2. `it`
3. `en_US`
4. `en`
5. The base class bundle

 **Tip:**

The `getBundle` method used to load the bundle looks for the default locale classes before it returns the base class bundle. If it fails to find a match, it throws a `MissingResourceException` error. A base class with no suffixes should always exist as a default. Otherwise, it may not find a match and the exception is thrown.

You must create a base resource bundle that contains all the text strings that are not part of the components themselves. This bundle should be in the default language of the application. You can create a resource bundle as a property file, as an XLIFF file, or as a Java class. After a resource bundle file has been created, you can edit the file using the Edit Resource Bundles dialog.

How to Create a Resource Bundle as a Property File or an XLIFF File

You can create a resource bundle as a property file or as an XLIFF file.

Before you begin:

It may help to understand what types of resource bundle you can create. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To create a resource bundle as a property file or an XLIFF file:

1. In the Applications window, right-click where you want to place the resource bundle file and choose **New > From Gallery**.

 **Note:**

If you are creating a localized version of the base resource bundle, save the file to the same directory as the base file.

2. In the New Gallery, select **General** and then **File**, and click **OK**.
3. In the Create File dialog, enter a name for the file using the convention `<name><_lang>.properties` for the using the properties file or `<name><_lang>.xlf` for using the XLIFF file, where the `<_lang>` suffix is provided for translated files, as in `_de` for German, and omitted for the base language.

 **Note:**

If you create a localized version of a base resource bundle, you must append the ISO 639 lowercase language code to the name of the file. For example, the Italian version of the `UIResources` bundle is `UIResources_it.properties`. You can add the ISO 3166 uppercase country code (for example `it_CH`, for Switzerland) if one language is used by more than one country. You can also add an optional nonstandard variant (for example, to provide platform or region information).

If you are creating the base resource bundle, do not append any codes.

4. Enter the content for the file. You can enter the content manually by entering the key-value pairs. You can use the Edit Resource Bundle dialog to enter the key-value pairs, as described in [How to Edit a Resource Bundle File](#).
 - If you are creating a property file, create a key and value for each string of static text for this bundle. The key is a unique identifier for the string. The value is the string of text in the language for the bundle. If you are creating a localized version of the base resource bundle, any key not found in this version inherits the values from the base class.

 **Note:**

All non-ASCII characters must be UNICODE-escaped or the encoding must be explicitly specified when compiling, for example:

```
javac -encoding ISO8859_5 UIResources_it.java
```

For example, the key and the value for the title of the myDemo page is:

```
myDemo.pageTitle=My Purchase Requests
```

- If you are creating an XLIFF file, enter the proper tags for each key-value pair. For example:

```
<?xml version="1.0" encoding="windows-1252" ?>
<xliff version="1.1" xmlns="urn:oasis:names:tc:xliff:document:1.1">
  <file source-language="en" original="myResources" datatype="xml">
    <body>
      <trans-unit id="NAME">
        <source>Name</source>
        <target/>
        <note>Name of employee</note>
      </trans-unit>
      <trans-unit id="HOME_ADDRESS">
        <source>Home Address</source>
        <target/>
        <note>Adress of employee</note>
      </trans-unit>
      <trans-unit id="OFFICE_ADDRESS">
        <source>Office Address</source>
        <target/>
        <note>Office building </note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

5. After you have entered all the values, click **OK**.

How to Create a Resource Bundle as a Java Class

You can create a resource bundle as a Java class.

Before you begin:

It may help to understand what types of resource bundle you can create. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To create a resource bundle as a Java class:

1. In the Applications window, right-click where you want the file to be placed and choose **New > From Gallery**.

Note:

If you are creating a localized version of the base resource bundle, it must reside in the same directory as the base file.

2. In the New Gallery, select **General** and then **Java Class**, and click **OK**.
3. In the Create Java Class dialog, enter a name and package for the class. The class must extend `java.util.ListResourceBundle`.

 **Note:**

If you are creating a localized version of a base resource bundle, you must append the ISO 639 lowercase language code to the name of the class. For example, the Italian version of the `UIResources` bundle might be `UIResources_it.java`. You can add the ISO 3166 uppercase country code (for example `it_CH`, for Switzerland) if one language is used by more than one country. You can also add an optional nonstandard variant (for example, to provide platform or region information).

If you are creating the base resource bundle, do not append any codes.

4. Implement the `getContents()` method, which returns an array of key-value pairs. Create the array of keys for the bundle with the appropriate values. Or use the Edit Resource Bundles dialog to automatically generate the code, as described in [How to Edit a Resource Bundle File](#). [Example 36-3](#) shows a base resource bundle Java class.

 **Note:**

Keys must be `String` objects. If you are creating a localized version of the base resource bundle, any key not found in this version will inherit the values from the base class.

Example 36-3 Base Resource Bundle Java Class

```
package sample;

import java.util.ListResourceBundle;

public class MyResources extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return contents;
    }

    static final Object[][] contents {
        {"button_Search", "Search"},
        {"button_Reset", "Reset"}
    };
}
```

How to Edit a Resource Bundle File

After you have created a resource bundle property file, XLIFF file, or Java class file, you can edit it using the source editor.

Before you begin:

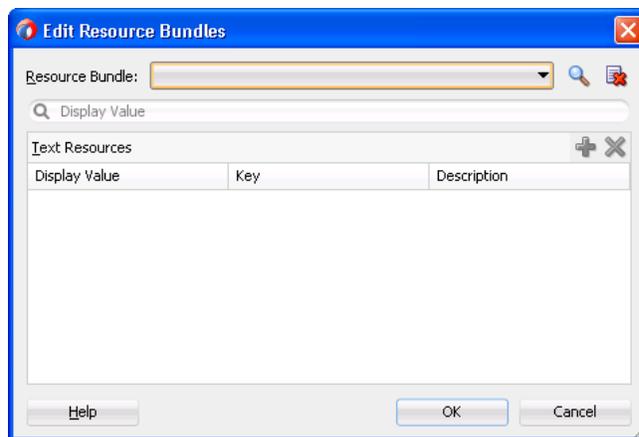
It may help to understand what types of resource bundles you can define and edit. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To edit a resource bundle after it has been created:

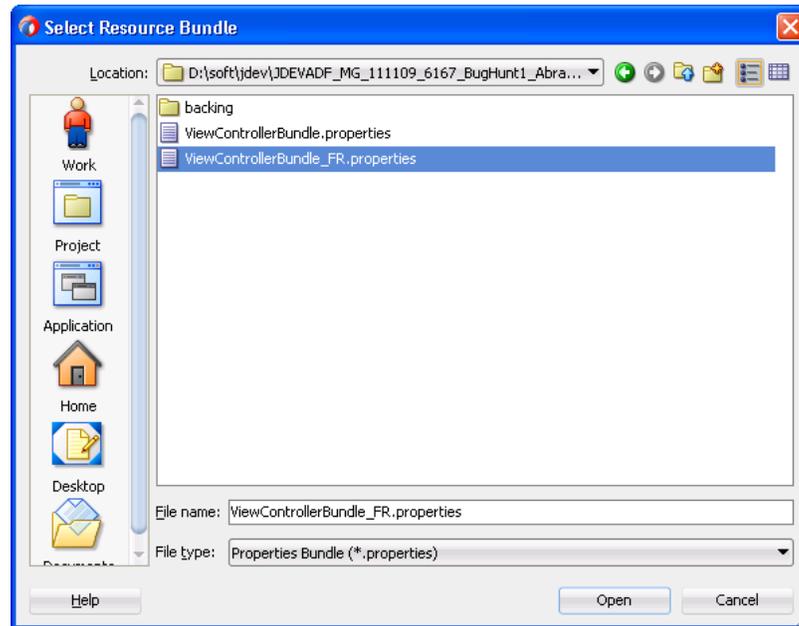
1. From the main menu, choose **Application > Edit Resource Bundles** from the main menu.
2. In the Edit Resource Bundles dialog, select the resource bundle file you want to edit from the **Resource Bundle** dropdown list, as shown in [Figure 36-3](#), or click the **Search** icon to launch the Select Resource Bundle dialog if the resource bundle file you want to edit does not appear in the **Resource Bundle** dropdown list.

Figure 36-3 Edit Resource Bundle Dialog



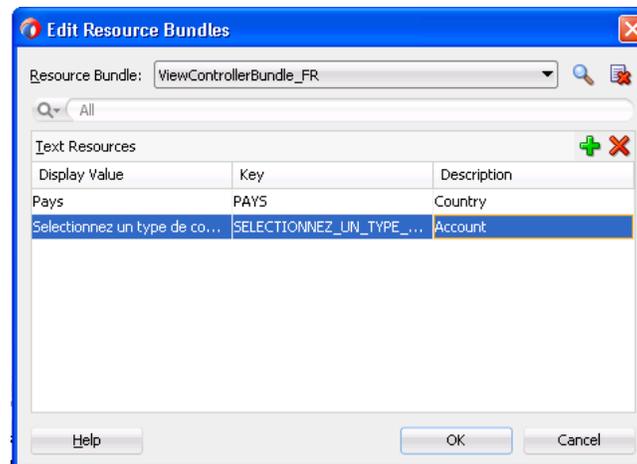
3. In the Select Resource Bundle dialog, select the file type from the **File type** dropdown list. Navigate to the resource bundle you want to edit, as shown in [Figure 36-4](#). Click **OK**.

Figure 36-4 Select Resource Bundle Dialog



4. In the Edit Resource Bundles dialog, click the **Add** icon to add a key-value pair, as shown in [Figure 36-5](#). When you have finished, click **OK**.

Figure 36-5 Adding Values to a Resource Bundle



How to Register a Locale for Your Application

You must register the locales that you want your application to support in the application's `faces-config.xml` file.

Before you begin:

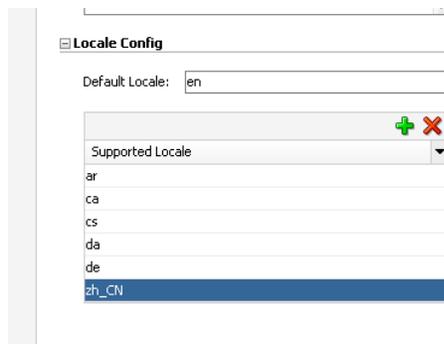
It may help to understand how you can manually manage resource bundles. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To register a locale for your application:

1. In the Applications window, expand the **WEB-INF** node and double-click **faces-config.xml**.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Application** navigation tab.
4. In the Application page, in the **Locale Config** section, click **Add** to add the code for the locale, as shown in [Figure 36-6](#).

Figure 36-6 Adding a Locale to faces-config.xml



After you have added the locales, the `faces-config.xml` file should have code similar to the following:

```
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>ar</supported-locale>
  <supported-locale>ca</supported-locale>
  <supported-locale>cs</supported-locale>
  <supported-locale>da</supported-locale>
  <supported-locale>de</supported-locale>
  <supported-locale>zh_CN</supported-locale>
</locale-config>
```

How to Register a Resource Bundle in Your Application

You must register the resource bundles that you want your application to use in the application's `faces-config.xml` file.

Before you begin:

It may help to understand how you can manually manage resource bundles. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To register a resource bundle:

1. In the Applications window, expand the **WEB-INF** node and double-click **faces-config.xml**.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Application** navigation tab.
4. In the Application page, in the **Resource Bundle** section, click **Add** and enter the fully qualified name of the base bundle that contains messages to be used by the application and a variable name that can be used to reference the bundle in an EL expression, as shown in [Figure 36-7](#).

Figure 36-7 Adding a Resource Bundle to faces-config.xml



After you have added the resource bundle, the `faces-config.xml` file should have code similar to the following:

```
<resource-bundle>
  <base-name>oracle.summit.SummitADF</base-name>
  <var>res</var>
</resource-bundle>
```

How to Use Resource Bundles in Your Application

You set your page encoding and response encoding to all supported languages and you bind to the resource bundle.

Before you begin:

It may help to understand how you can manually manage resource bundles. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To use a base resource bundle on your page:

1. If your page is a JSP document, set your page encoding and response encoding to be a superset of all supported languages. If no encoding is set, the page encoding defaults to the value of the response encoding set using the `contentType` attribute of the page directive. The following example shows the encoding for a sample page.

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
```

2. Bind all attributes that represent strings of static text displayed on the page to the appropriate key in the resource bundle, using the variable defined in the `faces-`

config.xml file for the <resource-bundle> element. The following example shows the code for the **View** button on the myDemo page.

```
<af:button text="#{res['myDemo.buttonbar.view']}"
    . . . />
```

Tip:

If you type the following syntax in the source editor, JDeveloper displays a dropdown list of the keys that resolve to strings in the resource bundle:

```
<af:button text="#{res.
```

JDeveloper completes the EL expression when you select a key from the dropdown list.

3. You can also use the `adfBundle` keyword to resolve resource strings from specific resource bundles as EL expressions in the JSF page.

The usage format is `#{adfBundle[bundleID] [resource_Key]}`, where *bundleID* is the fully qualified bundle ID, such as `project.EmpMsgBundle`, and *resource_Key* is the resource key in the bundle, such as `Deptno_LABEL`.

[Example 36-4](#) shows how `adfBundle` is used to provide the button text with a resource string from a specific resource bundle.

Example 36-4 Binding Using `adfBundle`

```
<af:button text="#{adfBundle['project.EmpMsgBundle'] ['Deptno_LABEL']}"
```

What You May Need to Know About ADF Skins and Control Hints

If you use an ADF skin and have created a custom resource bundle for the skin, you must also create localized versions of the resource bundle. Similarly, if your application uses control hints to set any text, you must create localized versions of the generated resource bundles for that text.

What You May Need to Know About Overriding a Resource Bundle in a Customizable Application

An override bundle is a resource bundle with key-value pairs that differ from the base resource bundle that you want to use in a customizable application that you develop using the Oracle Metadata Services (MDS) framework. If you create an override bundle, you need to configure your application's `adf-config.xml` file to support the overriding of the base resource bundle. For example, if you have a base resource bundle with the name `oracle.demo.CustAppUIBundle`, you configure an entry in your application's `adf-config.xml` file, as shown in [Example 36-5](#), to make it overrideable. Once it is marked as overridden, any customizations of that bundle will be stored in your customizable application's override bundle. The override bundle is maintained in both the default role in JDeveloper and in the customizable application. To avoid having to manage this override bundle in two locations, Oracle recommends that you:

- Create a separate resource bundle from the base resource bundle using JDeveloper's default role

- Add the key-value pairs that you want to override in your customizable application from the base resource bundle to this separate resource bundle

In the customizable application you pick these key-value pairs from the separate resource bundle. This makes sure that you do not create new key-value pairs in your customizable application's override bundle and that only the customizable application uses the override bundle.

For more information about the `adf-config.xml` file, see [Configuration in adf-config.xml](#). For more information about creating customizable applications using MDS, see the "Customizing Applications with MDS" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Example 36-5 Entry for Override Bundle in adf-config.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
  xmlns:config="http://xmlns.oracle.com/bc4j/configuration"
  xmlns:adf="http://xmlns.oracle.com/adf/config/properties">
  ...
<adf-resourcebundle-config xmlns="http://xmlns.oracle.com/adf/resourcebundle/config">
  <applicationBundleName>oracle/app.../xliffBundle/FusionAppsOverrideBundle</
applicationBundleName>
  <bundleList>
    <bundleId override="true">oracle.demo.CustAppUIBundle</bundleId>
  </bundleList>
</adf-resourcebundle-config>
</adf-config>
```

Configuring Pages for an End User to Specify Locale at Runtime

You can configure the ADF application to support end user selection of the locale-specific resource bundles at runtime.

You can configure an application so end users can specify the locale at runtime rather than the default behavior where the locale settings of the end user's browser determine the runtime locale. Implement this functionality if you want your application to allow end users to specify their preferred locale and save their preference.

How to Configure a Page for an End User to Specify Locale

Create a new page or open an existing page. Configure it so that:

- It references a backing bean to store locale information
- An end user can invoke a control at runtime to update the locale information in the backing bean
- The `locale` attribute of the `f:view` tag references the backing bean

Before you begin:

It may help to understand the configuration options available to you. For more information, see [Configuring Pages for an End User to Specify Locale at Runtime](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To configure a page for an end user to specify locale:

1. Create a page with a backing bean to store locale information.

For more information, see [How to Create JSF Pages](#).

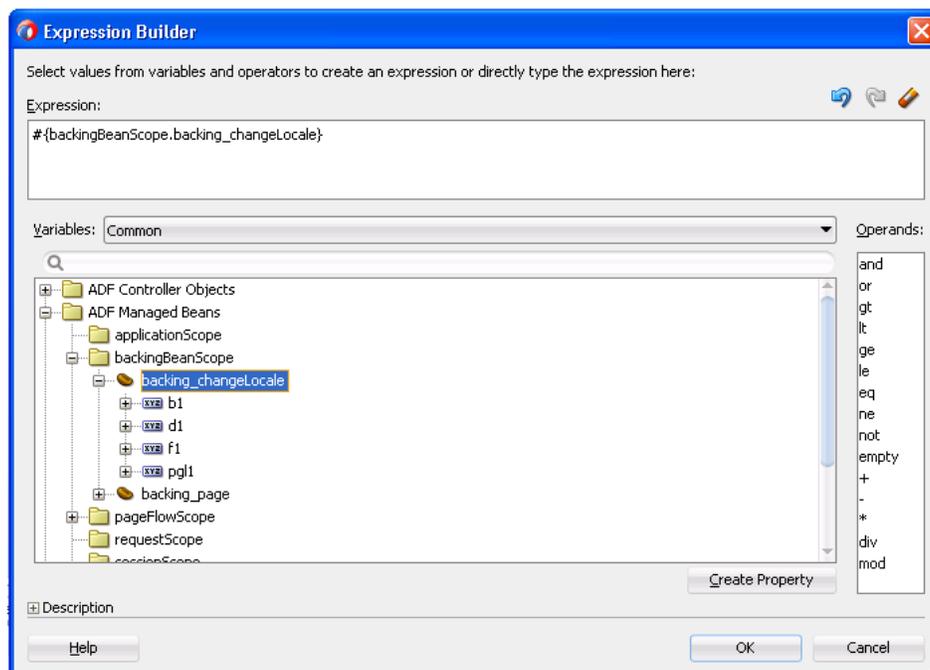
2. Provide a control (for example, a `selectOneChoice` component) that an end user can use to change locale.

For example, in the Components window, from the Text and Selection panel, drag a **Choice** component and drop it onto the page.

3. Bind the control to a backing bean that stores the locale value, as illustrated in the following example.

```
<af:selectOneChoice label="Select Locale"
  binding="#{backingBeanScope.backing_changeLocale.soc1}"
  id="soc1">
  <af:selectItem label="French" value="FR"
    binding="#{backingBeanScope.backing_changeLocale.sil}"
    id="sil"/>
  ...
</af:selectOneChoice>
```

4. Bind the `locale` attribute of the `f:view` tag to the locale value in the backing bean.
 - a. In the Structure window for the JSF page, right-click the `f:view` tag and choose **Go to Properties**.
 - b. In the Properties window, expand the **Common** section and choose **Expression Builder** from the context menu that appears when you click the icon that appears when you hover over the **Locale** field.
 - c. In the Expression Builder, bind to the locale value in the backing bean, as shown in [Figure 36-8](#).

Figure 36-8 Expression Builder Binding the Locale Attribute to a Backing Bean

5. Save the page.

What Happens When You Configure a Page to Specify Locale

JDeveloper generates a reference to the backing bean for the command component that you use to change the locale. [Example 36-6](#) shows an example using the `selectOneChoice` component. JDeveloper also generates the required methods in the backing bean for the page. [Example 36-7](#) shows extracts for the backing bean that correspond to [Example 36-6](#).

Example 36-6 selectOneChoice Component Referencing a Backing Bean

```
<af:selectOneChoice label="Select Locale"
  binding="#{backingBeanScope.backing_changeLocale.soc1}"
  id="soc1">
  <af:selectItem label="French" value="FR"
    binding="#{backingBeanScope.backing_changeLocale.sil}"
    id="sil"/>
  ...
</af:selectOneChoice>
```

Example 36-7 Backing Bean Methods to Change Locale

```
package view.backing;

...
import oracle.adf.view.rich.component.rich.input.RichSelectOneChoice;

public class ChangeLocale {
  ...
  ...
  private RichSelectOneChoice soc1;
  ...
}
```

```
...  
  
...  
public void setD2(RichDocument d2) {  
    this.d2 = d2;  
}  
  
...  
  
public void setSoc1(RichSelectOneChoice soc1) {  
    this.soc1 = soc1;  
}  
  
public RichSelectOneChoice getSoc1() {  
    return soc1;  
}  
  
public void setSil(RichSelectItem sil) {  
    this.sil = sil;  
}  
...  
}
```

What Happens at Runtime: How an End User Specifies a Locale

At runtime, an end user invokes the command component you configured to change the locale of the application. The backing bean stores the updated locale information. Pages where the `locale` attribute of the `f:view` tag reference the backing bean render using the locale specified by the end user.

The locale specified by the end user must be registered with your application. For more information about specifying a locale and associated resource bundles, see [How to Register a Locale for Your Application](#).

Configuring Optional ADF Faces Localization Properties

The ADF application provides properties that can assist in translation for a specific locale beyond text resources.

Along with providing text translation, ADF Faces also automatically provides other types of translation, such as currency codes and support for bidirectional rendering (also known as BiDi support). The application will automatically be displayed appropriately, based on the user's selected locale. However, you can also manually set the following localization settings for an application in the `trinidad-config.xml` file:

- `<currency-code>`: Defines the default ISO 4217 currency code used by `oracle.adf.view.faces.converter.NumberConverter` to format currency fields that do not specify a currency code in their own converter.
- `<number-grouping-separator>`: Defines the separator used for groups of numbers (for example, a comma). ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while it parses and formats.

- `<decimal-separator>`: Defines the separator used for the decimal point (for example, a period or a comma). ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. If set, this value is used by `oracle.adf.view.faces.converter.NumberConverter` while it parses and formats.
- `<right-to-left>`: Defines the direction in which text appears in a page. ADF Faces supports bidirectional rendering and automatically derives the rendering direction from the current locale, but you can explicitly set the default page rendering direction by using the values `true` or `false`.
- `<formatting-locale>`: Defines the date and number format appropriate to the selected locale. By default, ADF Faces will format dates and numbers in the same locale used for localized text. If you want dates and numbers formatted in a different locale, you can use an IANA-formatted locale (for example, `ja`, `fr-CA`). The contents of this element can also be an EL expression pointing at an IANA string or a `java.util.Locale` object.
- To set the time zone used for processing and displaying dates, and the year offset that should be used for parsing years with only two digits, use the following elements:
 - `<time-zone>`: By default, ADF Faces uses the time zone used by the application server if no value is set. If needed, you can use an EL expression that evaluates to a `TimeZone` object. This value is used by `org.apache.myfaces.trinidad.converter.DateTimeConverter` while converting strings to `Date`.
 - `<two-digit-year-start>`: This value is specified as a Gregorian calendar year and is used by `org.apache.myfaces.trinidad.converter.DateTimeConverter` to convert strings to `Date`. This element defaults to the year 1950 if no value is set. If needed, you can use a static integer value or an EL expression that evaluates to an `Integer` object.

For more information about the elements that you can configure in the `trinidad-config.xml` file, see [Configuration in trinidad-config.xml](#).

How to Configure Optional Localization Properties

You can configure optional localization properties by entering elements in your application's `trinidad-config.xml` file.

Before you begin:

It may help to understand what optional localization properties you can modify. For more information, see [Manually Defining Resource Bundles and Locales](#).

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see [Additional Functionality for Internationalizing and Localizing Pages](#).

To configure optional localization properties:

1. In the Applications window, expand the **WEB-INF** node and double-click **trinidad-config.xml**.

2. In the Trinidad Configuration Elements page of the Components window, drag the localization element that you want to configure and drop it inside the <trinidad-config> element.
3. Enter the desired value.

[Example 36-8](#) shows a sample `trinidad-config.xml` file with all the optional localization elements set.

Example 36-8 Configuring Currency Code and Separators for Numbers and Decimal Point

```
<!-- Set the currency code to US dollars. -->
<currency-code>USD</currency-code>

<!-- Set the number grouping separator to period for German -->
<!-- and comma for all other languages -->
<number-grouping-separator>
  #{view.locale.language=='de' ? '.' : ','}
</number-grouping-separator>

<!-- Set the decimal separator to comma for German -->
<!-- and period for all other languages -->
<decimal-separator>
  #{view.locale.language=='de' ? ',' : '.'}
</decimal-separator>

<!-- Render the page right-to-left for Arabic -->
<!-- and left-to-right for all other languages -->
<right-to-left>
  #{view.locale.language=='ar' ? 'true' : 'false'}
</right-to-left>

<formatting-locale>
  #{request.locale}
</formatting-locale>

<!-- Set the time zone to Pacific Daylight Savings Time -->
<time-zone>PDT</time-zone>
```

Developing Accessible ADF Faces Pages

This chapter describes how to add accessibility support to ADF Faces components by configuring `trinidad-config.xml` with the `<accessibility-mode>` and `<accessibility-profile>` elements. This chapter also describes accessibility guidelines for ADF pages that use partial page rendering, scripting, styles, and certain page and navigation structures.

This chapter includes the following sections:

- [About Accessibility Support In ADF Faces](#)
- [Specifying Component-Level Accessibility Properties](#)
- [Creating Accessible Pages](#)
- [Creating Accessible Active Data Components](#)
- [Running Accessibility Audit Rules](#)

About Accessibility Support In ADF Faces

ADF Faces components support creating user interfaces that comply with established accessibility guidelines for visually and physically impaired end users.

Accessibility involves making your application usable for persons with disabilities such as low vision or blindness, deafness, or other physical limitations. This means creating applications that can be used without a mouse (keyboard only), used with a screen reader for blind or low-vision users, and used without reliance on sound, color, or animation and timing.

ADF Faces user interface components have built-in accessibility support for visually and physically impaired users. User agents such as a web browser rendering to nonvisual media such as a screen reader can read component text descriptions to provide useful information to impaired users.

While the ADF Faces accessibility guidelines for components, page, and navigation structures is useful, it is not a substitute for familiarity with accessibility standards and performing accessibility testing with assistive technology.

Access key support provides an alternative method to access components and links using only the keyboard. ADF Faces accessibility audit rules provide direction to create accessible images, tables, frames, forms, error messages, and popup windows using accessible HTML markup. Additional framework and platform issues presented by client-side scripting, in particular using asynchronous JavaScript and XML (AJAX), have been addressed in Oracle's accessibility strategy.

Oracle software implements the U.S. Section 508 and Web Content Accessibility Guidelines (WCAG). The interpretation of these standards is available at <http://www.oracle.com/accessibility/standards.html>.

While the ADF Faces accessibility guidelines for components, page, and navigation structures is useful, it is not a substitute for familiarity with accessibility standards and

performing accessibility testing with assistive technology. While creating your application, make sure that the following criterion are met:

- Success criterion for sensory characteristics.
For more information, see the section on Success Criterion 1.3.3 [Sensory Characteristics] in <http://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html#contents>.
- Success criterion for images of text.
For more information, see the section on Success Criterion 1.4.5 [Images of Text] in <http://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html#contents>.
- Success criterion for visible focus.
For more information, see the section on Success Criterion 2.4.7 [Focus Visible] in <http://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html#contents>.
- Success criterion for elements receiving focus.
For more information, see the section on Success Criterion 3.2.1 [On Focus] in <http://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html#contents>.
- Success criterion for consistent identification of functional components
For more information, see the section on Success Criterion 3.2.4 [Consistent Identification] in <http://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html#contents>.

Additional Information for Accessibility Support in ADF Pages

You may also find it helpful to understand other ADF Faces features before you make your application accessible. Following are links to other features that work with accessibility.

- **Internationalization and localization:** The ADF skin that you create to apply to your application can be customized as part of a process to internationalize and localize ADF Faces pages. For more information about this process, see [Internationalizing and Localizing Pages](#).
- **Keyboard shortcuts:** Keyboard shortcuts provide an alternative to pointing devices for navigating the page. For more information about how to use keyboard shortcuts with accessibility, see [Keyboard Shortcuts](#).

Accessibility Support Guidelines at Sign-In

When developing an application, it is a good practice to provide options for users after application sign-in. The sign-in flow consists of two pages, as described here:

1. Sign-in page
2. Application home page

Note that the application may have additional authentication security set up and the product-specific home page.

Specifying Component-Level Accessibility Properties

ADF Faces components define a variety of properties that must be specified to make ADF applications accessible by visually or physically impaired end users.

Guidelines for component-specific accessibility are provided in [ADF Faces Component Accessibility Guidelines](#). The guidelines include a description of the relevant property with examples and tips. For information about auditing compliance with ADF Faces accessibility rules, see [Running Accessibility Audit Rules](#).

Access key support for ADF Faces input or command and go components such as `af:inputText`, `af:button`, and `af:link` involves defining labels and specifying keyboard shortcuts. While it is possible to use the tab key to move from one control to the next in a web application, keyboard shortcuts are more convenient and efficient.

To specify an access key for a component, set the component's `accessKey` attribute to a keyboard character (or mnemonic) that is used to gain quick access to the component. You can set the attribute in the Properties window or in the page source using `&` encoding.

The same access key can be bound to several components. If the same access key appears in multiple locations in the same page, the rendering agent will cycle among the components accessed by the same key. That is, each time the access key is pressed, the focus will move from component to component. When the last component is reached, the focus will return to the first component.

Using access keys on `af:button` and `af:link` components may immediately activate them in some browsers. Depending on the browser, if the same access key is assigned to two or more go components on a page, the browser may activate the first component instead of cycling through the components that are accessed by the same key.

To develop accessible page and navigation structures, follow the additional accessibility guidelines described in [Creating Accessible Pages](#).

ADF Faces Component Accessibility Guidelines

To develop accessible ADF Faces components, follow the guidelines described in the component's tag documentation, and in [Table 37-1](#). Components not listed in [Table 37-1](#) do not have accessibility guidelines.

Note:

In cases where the `label` property is referenced in the accessibility guidelines, the `labelAndAccessKey` property may be used where available, and is the preferred option.

Unless noted otherwise, you can also label ADF Faces input and select controls by:

- Specifying the `for` property in an `af:outputLabel` component
- Specifying the `for` property in an `af:panelLabelAndMessage` component

Table 37-1 ADF Faces Components Accessibility Guidelines

Component	Guidelines
af:activeImage	Specify the shortDesc property. If the image is only present for decorative purposes and communicates no information, set shortDesc to the empty string.
af:chooseColor	For every af:chooseColor component, there must be at least one af:inputColor component with a chooseId property, which points to the af:chooseColor component.
af:chooseDate	For every af:chooseDate component, there must be at least one af:inputDate component with a chooseId property, which points to the af:chooseDate component.
af:column	Specify the headerText property, or provide a header facet. In a table, you must identify at least one column component as a row header by setting rowHeader to true or unstyled. Ensure that the rowHeader column provides a unique textual value, and columns that contain an input component are not assigned as the row header. For every child input component, set the Simple attribute to true, and ensure that the input component has a label assigned. The label is not displayed, but is present on the page so that it can be read by screen reader software. If you wish to provide help information for the column, use helpTopicId. If you use a filter facet to set a filter on a column, ensure that the filter component has a label assigned.
af:button af:link	One of the following properties must be specified: text, textAndAccessKey, or shortDesc (used in conjunction with icon). The text should specify the action or destination activated by the component and make sense when read out of context. For example use "go to index" instead of "click here". If present, the text or textAndAccessKey property is used as the label for the component. For an icon only button or link, the shortDesc labels the component. Unique buttons or links must have unique text.
af:commandMenuItem af:commandNavigationItem	One of the following properties must be specified: text, textAndAccessKey, or shortDesc. Usually, the text or textAndAccessKey property is used as the label for the component. Unique buttons and links must have unique text.
af:dialog	Specify the title property.
af:document	If you wish to provide help information, use helpTopicId.

Table 37-1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
af:goMenuItem	<p>Specify the <code>text</code> property. The text should specify where the link will take the user and make sense when read out of context. For example use "go to index" instead of "click here." Multiple links that go to the same location must use the same text and unique links must have unique text.</p> <p>Usually, the <code>text</code> or <code>textAndAccessKey</code> property is used as the label for the component. Unique buttons and links must have unique text.</p>
af:icon	Specify the <code>shortDesc</code> property. If the icon is only present only for decorative purposes and communicates no information, set <code>shortDesc</code> to the empty string.
af:image	<p>Specify the <code>shortDesc</code> property. If the image is only present for decorative purposes and communicates no information, set <code>shortDesc</code> to the empty string.</p> <p>Use the <code>longDescURL</code> property for images where a complex explanation is necessary. For example, charts and graphs require a description file that includes all the details that make up the chart.</p>
af:inlineFrame	Specify the <code>shortDesc</code> property.
af:inputColor	Specify the <code>label</code> property.
af:inputComboboxListOfValues	For <code>af:inputComboboxListOfValues</code> and <code>af:inputListOfValues</code> components, the <code>searchDesc</code> must also be specified.
af:inputDate	
af:inputFile	If you wish to provide help information, use <code>helpTopicId</code> .
af:inputListOfValues	
af:inputNumberSlider	
af:inputNumberSpinbox	
af:inputRangeSlider	
af:inputText	
af:outputFormatted	The <code>value</code> property must specify valid HTML.
af:outputLabel	Specify the <code>value</code> or <code>valueAndAccessKey</code> property.
af:media	<p>When including multimedia content in your application, make sure that all accessibility requirements for that media are met.</p> <p>Here are some examples:</p> <ul style="list-style-type: none"> • If including audio only content, provide an alternate way of conveying the same information, such as a textual transcript of the audio. • Any audio that plays automatically for more than 3 seconds must have a pause, stop, or mute control. • If including video only content, provide an alternate way of conveying the same information, such as a textual or audio transcript. • When including video content with audio, captions (and audio descriptions) should also be available.

Table 37-1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
af:panelBox	Specify the text property.
af:panelHeader	If you wish to provide help information, use helpTopicId.
af:panelLabelAndMessage	Specify the label or labelAndAccessKey property. When using this component to label an ADF Faces input or select control, the for property must be specified. If you wish to provide help information, use helpTopicId.
af:panelSplitter	Refer to How to Use Page Structures and Navigation .
af:panelStretchLayout	
af:panelWindow	Specify the title property. If you wish to provide help information, use helpTopicId.
af:poll	When using polling to update content, allow end users to control the interval, or to explicitly initiate updates instead of polling.
af:query	Specify the following properties: <ul style="list-style-type: none"> headerText addFieldsButtonAccessKey addFieldsButtonText resetButtonAccessKey resetButtonText saveButtonAccessKey saveButtonText searchButtonAccessKey searchButtonText If you wish to provide help information, use helpTopicId.
af:quickQuery	Specify the label and searchDesc properties. If you wish to provide help information, use helpTopicId.
af:region	If you wish to provide help information, use helpTopicId.
af:richTextEditor	Specify the label property. If you wish to provide help information, use helpTopicId.
af:selectBooleanCheckbox	One of the following properties must be specified: text, textAndAccessKey, or label.
af:selectBooleanRadio	If you wish to provide help information, use helpTopicId.
af:selectItem	Specify the label property. Note that using the for attribute of af:outputLabel and af:panelMessageAndLabel components is not an acceptable alternative.

Table 37-1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
<code>af:selectManyCheckbox</code>	Specify the <code>label</code> property.
<code>af:selectManyChoice</code>	For the <code>af:selectManyShuttle</code> and <code>af:selectOrderShuttle</code> components, the <code>leadingHeader</code> and <code>trailingHeader</code> properties must be specified.
<code>af:selectManyListbox</code>	
<code>af:selectManyShuttle</code>	
<code>af:selectOneChoice</code>	
<code>af:selectOneListbox</code>	
<code>af:selectOneRadio</code>	
<code>af:selectOrderShuttle</code>	If you wish to provide help information, use <code>helpTopicId</code> .
<code>af:showDetailHeader</code>	Specify the <code>text</code> property. If you wish to provide help information, use <code>helpTopicId</code> .
<code>af:showDetailItem</code>	One of the following properties must be specified: <code>text</code> , <code>textAndAccessKey</code> , or <code>shortDesc</code> .
<code>af:showPopupBehavior</code>	Specify the <code>triggerType</code> property. For all popups that are accessible with the mouse, make sure they are also through the keyboard. For example, if you want to display a note window on <code>mouseover</code> , set up two <code>showPopupBehavior</code> tags: one triggering on <code>mouseover</code> and one triggering on <code>action</code> , so that a keyboard only can also access the popup. Do not show a popup on <code>focus</code> or <code>blur</code> event types. It makes application difficult to use by a keyboard-only user. While these are available event types, they should not be used in any real application with <code>showPopupBehavior</code> .
<code>af:table</code>	Specify the <code>summary</code> property. The summary should describe the purpose of the table.
<code>af:treeTable</code>	All table columns must have column headers. If a label is specified for an input component inside <code>af:column</code> , ADF Faces automatically inserts row information into the application-specified label of the input component. Typically, the application-specified label matches the column header text, and along with the inserted row information it provides a unique identity for each input component.

Using ADF Faces Table Components with a Screen Reader

If you are using ADF Faces table components in your web application, you must designate a column (or columns) that best describe the row information as the row header columns. The row header columns are used by screen reader software to announce the row when the end user selects it.

Sometimes, for display purposes, you may not want to have a row header. In such a case, you must define one or more columns in the table to have the `rowHeader` attribute set to `unstyled`. The `unstyled` row header columns do not display differently than other columns, but are properly identified as the row headers for accessibility.

ADF Data Visualization Components Accessibility Guidelines

To develop accessible ADF Data Visualization Components, follow the accessibility guidelines described in [Table 37-2](#). Components not listed do not have accessibility guidelines.

Table 37-2 ADF Data Visualization Components Accessibility Guidelines

Component	Guideline
dvt:projectGantt dvt:resourceUtilizationGantt dvt:schedulingGantt	Specify the <code>summary</code> property. The summary should describe the purpose of the Gantt chart component.
dvt:areaChart dvt:barChart dvt:bubbleChart dvt:comboChart dvt:funnelChart dvt:horizontalLineChart	Charts automatically assign a value to the <code>shortDesc</code> property for the following chart child components when you add them to a page. The property describes the series, group, and value information for the component. <ul style="list-style-type: none"> <code>dvt:chartDataItem</code> <code>dvt:funnelDataItem</code> <code>dvt:pieDataItem</code>
dvt:lineChart dvt:pieChart dvt:scatterChart	Specify the <code>shortDesc</code> property for the following chart reference objects when used. The property should describe the purpose and data information of the component. <ul style="list-style-type: none"> <code>dvt:referenceArea</code> <code>dvt:referenceLine</code>
dvt:diagram	Specify the <code>summary</code> property. The summary should describe the purpose of the component. Also specify the <code>shortDesc</code> property for the <code>dvt:diagramNode</code> and <code>dvt:diagramLinkchild</code> components. The property should contain information identifying the label and node data for the node.
dvt:dialGauge dvt:ledGauge dvt:ratingGauge dvt:statusMeterGauge	Specify the <code>shortDesc</code> property. The property should describe the purpose and data information of the gauge.
dvt:hierarchyViewer	Specify the <code>summary</code> property. The summary should describe the purpose of the component.
dvt:map	Specify the <code>summary</code> property. The summary should describe the purpose of the geographic map.
dvt:nBoxNode	Specify the <code>shortdesc</code> property. The property should describe the purpose and data of the node.
dvt:pivotTable	Specify the <code>summary</code> property. The summary should describe the purpose of the timeline component.
dvt:sparkChart	Specify the <code>shortDesc</code> property. The property should describe the purpose and data of the spark chart component.
dvt:stockGraph	Specify the <code>shortDesc</code> property. The <code>shortDesc</code> property should describe the purpose of the graph.

Table 37-2 (Cont.) ADF Data Visualization Components Accessibility Guidelines

Component	Guideline
dvt:sunburst	Specify the <code>summary</code> property. The summary should describe the purpose of the component. Also specify the <code>shortDesc</code> property for the <code>dvt:sunburstNode</code> child component. The property should contain information identifying the label and node data for the node.
dvt:thematicMap	Specify the <code>summary</code> property. The summary should describe the purpose of the thematic map.
dvt:timeline	Specify the <code>summary</code> property. The summary should describe the purpose of the pivot table component.
dvt:treemap	Specify the <code>summary</code> property. The summary should describe the purpose of the component. Also specify the <code>shortDesc</code> property for the <code>dvt:treemapNode</code> child components. The property should contain information identifying the label and node data for the node.

 **Note:**

DVT chart, sunburst, and treemap components conform to the Accessible Rich Internet Applications Suite (WAI-ARIA) technical specification. The WAI-ARIA framework defines roles, states, and properties to make widgets, navigation, and behaviors accessible. For more information about WAI-ARIA, see <http://www.w3.org/WAI/intro/aria>.

How to Define Access Keys for an ADF Faces Component

In the Properties window of the component for which you are defining an access key, enter the mnemonic character in the `accessKey` attribute field. When simultaneously setting the text, label, or value and mnemonic character, use the ampersand (&) character in front of the mnemonic character in the relevant attribute field.

Before you begin:

It may be helpful to have an understanding of component-level accessibility guidelines. For more information, see [Specifying Component-Level Accessibility Properties](#). You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Information for Accessibility Support in ADF Pages](#).

Defining Access Keys

Use one of four attributes to specify a keyboard character for an ADF Faces input or command and go component:

- `accessKey`: Use to set the mnemonic character used to gain quick access to the component. For command and go components, the character specified by this attribute must exist in the text attribute of the instance component; otherwise, ADF

Faces does not display the visual indication that the component has an access key.

The following example shows the code that sets the access key to the letter `h` for the `af:link` component. In Internet Explorer, when the user presses the keys `ALT+H`, the text value of the component will be brought into focus. Note that the `ALT` key might not act as the access key in every web browser. Refer to the browser's documentation to know the shortcut keys of the browser.

```
<af:link text="Home" accessKey="h">
```

- `textAndAccessKey`: Use to simultaneously set the text and the mnemonic character for a component using the ampersand (`&`) character. In JSPX files, the conventional ampersand notation is `&`. In JSP files, the ampersand notation is simply `&`. In the Properties window, you need only the `&` character.

The following example shows the code that specifies the button text as `Home` and sets the access key to `H`, the letter immediately after the ampersand character, for the `af:button` component.

```
<af:button textAndAccessKey="&amp;Home"/>
```

- `labelAndAccessKey`: Use to simultaneously set the `label` attribute and the access key on an input component, using conventional ampersand notation.

The following example shows the code that specifies the label as `Date` and sets the access key to `a`, the letter immediately after the ampersand character, for the `af:selectInputDate` component.

```
<af:inputSelectDate value="Choose date" labelAndAccessKey="D&amp;ate"/>
```

- `valueAndAccessKey`: Use to simultaneously set the `value` attribute and the access key, using conventional ampersand notation.

The following example shows the code that specifies the label as `Select Date` and sets the access key to `e`, the letter immediately after the ampersand character, for the `af:outputLabel` component.

```
<af:outputLabel for="someid" valueAndAccessKey="Select Dat&amp;e"/>  
<af:inputText simple="true" id="someid"/>
```

Access key modifiers are browser and platform-specific. If you assign an access key that is already defined as a menu shortcut in the browser, the ADF Faces component access key will take precedence. Refer to your specific browser's documentation for details.

In some browsers, if you use a space as the access key, you must provide the user with the information that `Alt+Space` or `Alt+Spacebar` is the access key because there is no way to present a blank space visually in the component's label or textual label. For that browser you could provide text in a component tooltip using the `shortDesc` attribute.

How to Define Localized Labels and Access Keys

Labels and access keys that must be displayed in different languages can be stored in resource bundles where different language versions can be displayed as needed. Using the `<resource-bundle>` element in the JSF configuration file available in JSF 1.2, you can make resource bundles available to all the pages in your application without using a `af:loadBundle` tag in every page.

Before you begin:

It may be helpful to have an understanding of component-level accessibility guidelines. For more information, see [Specifying Component-Level Accessibility Properties](#). You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Information for Accessibility Support in ADF Pages](#).

To define localized labels and access keys:

1. Create the resource bundles as simple `.properties` files to hold each language version of the labels and access keys. For details, see [How to Create a Resource Bundle as a Property File or an XLIFF File](#).
2. Add a `<locale-config>` element to the `faces-config.xml` file to define the default and supported locales for your application. For details, see [How to Register a Locale for Your Application](#).
3. Create a key and value for each string of static text for each resource bundle. The key is a unique identifier for the string. The value is the string of text in the language for the bundle. In each value, place an ampersand (& or `&`) in front of the letter you wish to define as an access key.

For example, the following code defines a label and access key for an edit button field in the `UIStrings.properties` base resource bundle as `Edit`:

```
srlist.buttonbar.edit=&Edit
```

In the Italian language resource bundle, `UIStrings_it.properties`, the following code provides the translated label and access key as `Aggiorna`:

```
srlist.buttonbar.edit=A&aggiorna
```

4. Add a `<resource-bundle>` element to the `faces-config.xml` file for your application. The following example shows an entry in a JSF configuration file for a resource bundle.

```
<resource-bundle>  
  <var>res</var>  
  <base-name>resources.UIStrings</base-name>  
</resource-bundle>
```

Once you set up your application to use resource bundles, the resource bundle keys show up in the Expression Language (EL) editor so that you can assign them declaratively.

In the following example, the UI component accesses the resource bundle:

```
<af:outputText value="#{res['login.date']}"/
```

For more information, see [Internationalizing and Localizing Pages](#).

Creating Accessible Pages

Certain features of ADF Faces pages, such as partial page rendering or page navigation, must be considered to make ADF applications accessible by visually or physically impaired end users.

In addition to component-level accessibility guidelines, you should also follow page-level accessibility guidelines when you design your application. While component-level guidelines may determine how you use a component, page-level accessibility

guidelines are more involved with the overall design and function of the application as a whole.

The page-level accessibility guidelines are for:

- Using partial page rendering
- Using scripting
- Using styles
- Using page structures and navigation
- Using WAI-ARIA landmark regions

When designing the application pages, you must follow these general accessibility guidelines described in [Table 37-3](#).

Table 37-3 General Accessibility Guidelines

Guideline	Action
Avoid using raw HTML content	If possible, avoid using raw HTML content. If raw HTML content is required, use <code>af:outputFormatted</code> and ensure that the content is valid.
Use the clearest and simplest language appropriate for a site's content	Ensure language clarity and simplicity across the application.
Provide keyboard alternatives to drag and drop	Any functionality that uses drag and drop operations must also be exposed through a keyboard-accessible interface, such as Cut, Copy, and Paste menu items.
Review accessibility standards	You must be aware of relevant accessibility standards, such as the Web Content Accessibility Guidelines. Although the ADF Faces framework and components hide many of the implementation details, you should be familiar with these guidelines.
Write text that describes the link's purpose	Ensure that the purpose of each link can be determined from the link text alone, or from the link text together with its programmatically determined link context, except where the purpose of the link would be ambiguous to users in general.
Provide information about the general layout of the site, such as a site map or table of contents	Ensure that site layout requirements are met.
Provide multiple ways to locate a page	Ensure that page access requirements are met across the application. Pages that are the result of a process, or a step in a process, can be excluded.
Provide visual separation between adjacent links	Ensure that adjacent links are visually separated, and that a single link containing white space does not appear as multiple links.
Provide accessibility support for non-ADF content	Ensure that non-ADF Faces content in the page is accessible. The content can come from other Oracle products, or any third-party products.
Provide accessibility support for external documents	Ensure that external documents, such as Word documents and PDF files, are accessible. The documents could be generated by the product, or be shipped with the product, and must have least one accessible version.

The guidelines described in this section, and its subsections, follow Oracle Global HTML Accessibility Guidelines, which combines the guidelines of Section 508 and Web Content Accessibility Guidelines. ADF Faces components ease your responsibility, as they implicitly meet several accessibility guidelines. For example, ADF Faces renders the `lang` attribute on every page, and all headers rendered by ADF Faces components use the appropriate HTML header elements.

How to Use Partial Page Rendering

Screen readers do not reread the full page in a partial page request. When using Partial Page Rendering (PPR), you must follow the guidelines described in [Table 37-4](#).

Table 37-4 Partial Page Rendering Guidelines for Accessibility

Guideline	Action
Prefer downstream partial page changes	<p>Partial page rendering causes the screen reader software to read the page starting from the component that triggered the partial action. Therefore, place the target component after the component that triggers the partial request; otherwise, the screen reader software will not read the updated target.</p> <p>For example, the most common PPR use case is the master-detail user interface, where selecting a value in the master component results in partial page replacement of the detail component. In such scenarios, the master component must always appear before the detail component in the document order.</p>
Provide guidance for partial page changes	<p>Screen reader or screen magnifier users may have difficulty determining exactly what content has changed as a result of partial page rendering activity. It may be helpful to provide guidance in the form of inline text descriptions that identify relationships between key components in the page. For example, in a master-detail scenario, inline text might explain that when a row on master component is updated, the detail component is also updated. Alternatively, a help topic might explain the structure in the page and the relationships between components.</p>

How to Use Scripting

Client-side scripting should not be used for any application problem for which there is a declarative solution and so should be kept to a minimum.

When using scripting, you must follow these guidelines as described in [Table 37-5](#).

Table 37-5 Scripting Guidelines for Accessibility

Guideline	Action
Keep scripting to a minimum	Avoid client-side scripting.
Do not interact with the component Document Object Model (DOM) directly	ADF Faces components automatically synchronize with the screen reader when DOM changes are made. Direct interaction with the DOM is not allowed.

Table 37-5 (Cont.) Scripting Guidelines for Accessibility

Guideline	Action
Do not use JavaScript timeouts	Screen readers do not reliably track modifications made in response to timeouts implemented using the JavaScript <code>setTimeout()</code> or <code>setInterval()</code> APIs. Do not call these methods.
Provide keyboard equivalents	Some users may not have access to the mouse input device. For example, some users may be limited to keyboard use only, or may use alternate input devices or technology such as voice recognition software. When adding functions using client-side listeners, ensure that the function is accessible independent in device. Practically speaking this means that: <ul style="list-style-type: none"> • All functions must be accessible using the keyboard events. • Click events should be preferred over mouse-over or mouse-out. • Mouse-over or mouse-out events should additionally be available through the click event.
Avoid focus changes	Focus changes can be confusing to screen reader users as they involve a change of context. Design your application to avoid changing the focus programmatically, especially in response to focus events. Additionally, do not set popup windows to be displayed in response to focus changes because standard tabbing is disrupted.
Provide explicit popup triggers	Screen readers do not automatically respond to inline popup startups. To force the screen reader software to read the popup contents, the ADF Faces framework explicitly moves the keyboard focus to any popup window just after it is opened. An explicit popup trigger such as a link or button must be provided, or the same information must be available in some other keyboard or screen reader accessible way.
Provide text description for embedded objects	Ensure that each embedded object has a proper text description associated with it. The <code>OBJECT</code> element must specify the <code>title</code> attribute; the <code>APPLET</code> element must specify the <code>alt</code> attribute. Run the audit report to verify the audit rule for <code>af:media</code> .
Provide links to download required plug-ins	ADF Faces does not make use of any plug-ins such as Java, Flash, or PDF. You must ensure that the appropriate links are provided for plug-ins required by the application.
Provide accessible content for plug-ins	Ensure that all content conveyed by applets and plug-ins is accessible, or provide an alternate means of accessing equivalent content.
Avoid input-device dependency for event handlers	Ensure that event handlers are input device-independent, except for events not essential to content comprehension or application operation, such as mouse rollover image swaps.

In addition to scripting guidelines, you must also provide some programming guidelines. Many of these guidelines are implicitly adopted by ADF Faces and no action is required to implement them. The programming guidelines are listed in [Table 37-6](#).

Table 37-6 Application Programming Guidelines for Accessibility

Guideline	Action
Avoid using markup to redirect pages	No action required. ADF Faces does not use markup to redirect pages.
Specify the <code>DOCTYPE</code> of each page	No action required. ADF Faces specifies the <code>DOCTYPE</code> for every page.
Avoid using ASCII characters to render drawings or figures	Ensure that no ASCII art is included in the application.
Avoid disrupting the features of the platform that are defined, in the documentation intended for application developers, as having an accessibility usage	No action required. ADF Faces ensures that content generated by the ADF Faces components does not disrupt platform accessibility features.
Describe components that control the appearance of other components	Ensure that ADF Faces components that control other components have proper descriptions. The control over other components may include enabling or disabling, hiding or showing, or changing the default values of other controls.
Always use well-formed HTML code	No action required. ADF Faces is responsible for ensuring that its components generate well-formed HTML code.
Do not use deprecated HTML elements	No action required. ADF Faces is responsible for ensuring that its components do not use deprecated HTML elements.
Ensure that section headings are self-explanatory, and use header elements <code>H1</code> through <code>H6</code>	No action required. All headers rendered by ADF Faces components use the appropriate HTML header elements.
Ensure that the list content uses appropriate HTML list elements	No action required. All lists rendered by ADF Faces components use the appropriate HTML list elements, such as <code>OL</code> , <code>UL</code> , <code>LI</code> , <code>DL</code> , <code>DT</code> , and <code>DD</code> .
Mark quotations with proper elements	Ensure that quotations are appropriately marked up using <code>Q</code> or <code>BLOCKQUOTE</code> elements. Do not use quotation markup for formatting effects such as indentation.
Identify the primary natural language of each page with the <code>lang</code> attribute on the HTML element	No action required. ADF Faces renders the <code>lang</code> attribute on every page.
Ensure that all form elements have a label associated with them using markup	Run the audit report. The <code>Verify that the component is labeled</code> audit rule warns about missing labels.
Provide unique titles to each <code>FRAME</code> or <code>IFRAME</code> elements	Run the audit report. The <code>Verify that the component has a short description</code> audit rule warns when <code>af:inlineFrame</code> is missing the <code>shortDesc</code> title. Note that ADF Faces <code>af:inlineFrame</code> does not provide access to <code>longDesc</code> .

Table 37-6 (Cont.) Application Programming Guidelines for Accessibility

Guideline	Action
Provide a title to each page of the frame	Run the audit report. The <code>Verify</code> that the component has a <code>title</code> audit rule warns when <code>af:document</code> is missing the <code>title</code> attribute.
Ensure that popup windows have focus when they open, and focus must return to a logical place when the popup window is closed	Popup windows provided by ADF Faces components always appear in response to explicit user action. ADF Faces also ensures that focus is properly moved to the popup window on launch and restored on dismiss. However, for popup windows which are launched manually through <code>af:clientListener</code> or <code>af:showPopupBehavior</code> , you must ensure that the pop-up window is launched in response to explicit user action.

How to Use Styles

ADF Faces components are already styled and you may not need to make any changes. When using cascading style sheets (CSS) to directly modify the default appearance of ADF Faces components, you must follow the guidelines as described in [Table 37-7](#).

Table 37-7 Style Guidelines for Accessibility

Guideline	Action
Keep CSS use to a minimum	You are not required to specify CSS directly to the ADF components, as they are already styled.
Do not override default component appearance	Be aware of accessibility implications when you override default component appearance. Using CSS to change the appearance of components can have accessibility implications. For example, changing colors may result in color contrast issues.
Use scalable size units	When specifying sizes using CSS, use size units that scale relative to the font size rather than absolute units. For example, use <code>em</code> , <code>ex</code> or <code>%</code> units rather than <code>px</code> . This is particularly important when specifying heights using CSS, because low-vision users may scale up the font size, causing contents restricted to fixed or absolute heights to be clipped.
Do not use CSS positioning	Use CSS positioning only in the case of positioning the stretched layout component. Do not use CSS positioning elsewhere.
Use style sheets to change the layout and presentation of the screen	No action required. ADF Faces uses structural elements with style sheets to implement layout.
Create a style of presentation that is consistent across pages	No action required. ADF Faces provides a consistent style of presentation via its skinning architecture.
Do not use colors or font styles to convey information or indicate an action	Ensure that colors, or font styles, are not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.

How to Use Page Structures and Navigation

When using page structures and navigation tools, you must follow the guidelines as described in [Table 37-8](#).

Table 37-8 Style Guidelines for Page Structures and Navigation

Guideline	Action
Use <code>af:panelSplitter</code> for layouts	<p>When implementing geometry-managed layouts, using <code>af:panelSplitter</code> allows users to:</p> <ul style="list-style-type: none"> • Redistribute space to meet their needs • Hide or collapse content that is not of immediate interest. <p>If you are planning to use <code>af:panelStretchLayout</code>, you should consider using <code>af:panelStretchLayout</code> instead when appropriate.</p> <p>These page structure qualities are useful to all users, and are particularly helpful for low-vision users and screen-reader users</p> <p>As an example, a chrome navigation bar at the top of the page should be placed within the first facet of a vertical <code>af:panelSplitter</code> component, rather than within the top facet of <code>af:panelStretchLayout</code> component. This allows the user to decrease the amount of space used by the bar, or to hide it altogether. Similarly, in layouts that contain left, center, or right panes, use horizontal splitters to lay out the panes.</p>
Enable scrolling of flow layout contents	<p>When nesting flow layout contents (for example layout controls inside of geometry-managed parent components such as <code>af:panelSplitter</code> or <code>af:panelStretchLayout</code>), wrap <code>af:panelGroupLayout</code> with <code>layout="scroll"</code> around the flow layout contents. This provides scrollbars in the event that the font size is scaled up such that the content no longer fits. Failure to do this can result in content being clipped or truncated.</p>
Use header based components to identify page structure	<p>HTML header elements play an important role in screen readability. Screen readers typically allow users to gain an understanding of the overall structure of the page by examining or navigating across HTML headers. Identify major portions of the page through components that render HTML header contents including:</p> <ul style="list-style-type: none"> • <code>af:panelHeader</code> • <code>af:showDetailHeader</code> • <code>af:showDetailItem</code> in <code>af:panelAccordion</code> (each accordion in a pane renders an HTML header for the title area)
Use <code>af:breadcrumbs</code> to identify page location	<p>Accessibility standards require that users be able to determine their location within a web site or application. The use of <code>af:breadcrumbs</code> achieves this purpose.</p>
Use <code>af:skipLinkTarget</code> to provide a skip link target	<p>The <code>af:skipLinkTarget</code> tag provides a way to automatically generate a skip link at the beginning of the page. This is helpful for both screen reader and keyboard users, who benefit from the ability to skip over page-level chrome that is repeated on all pages. The <code>af:skipLinkTarget</code> tag should be specified once per page template.</p>

Table 37-8 (Cont.) Style Guidelines for Page Structures and Navigation

Guideline	Action
Maintain consistency for navigational mechanisms that are repeated on multiple pages	Ensure navigation consistency by using the ADF Faces navigation components.
Provide a method for skipping repetitive content	If repetitive content (including navigation links) is provided at the top of a page, ensure that the <code>af:skipLinkTarget</code> is used to skip over the repetitive content.

How to Use Images and Tables

When using images, you must follow the guidelines as described in [Table 37-9](#).

Table 37-9 Style Guidelines for Images

Guideline	Action
Specify description in <code>alt</code> attribute of non-decorative images	Run the audit report. The <code>Verify that the component has a short description</code> audit rule warns about missing <code>shortDesc</code> attributes. Ensure that <code>shortDesc</code> value is meaningful.
Ensure that decorative images, such as spacer images, specify an <code>alt=""</code> attribute	Run the audit report. The <code>Verify that the component has a short description</code> audit rule warns about missing <code>shortDesc</code> attributes. Ensure that <code>shortDesc</code> value is meaningful.
Specify description in <code>alt</code> attribute of complex images, such as charts	Ensure that the <code>longDesc</code> attribute is specified for complex <code>af:image</code> components.
Provide audio or text alternative for prerecorded synchronized media, such as videos	Ensure that the appropriate audio or text alternatives are provided.
Provide captions for prerecorded synchronized media	Ensure that the appropriate captions are provided. Captions are not required if the synchronized media is an alternative to text and is clearly labeled.

When using tables, you must follow the guidelines as described in [Table 37-10](#).

Table 37-10 Style Guidelines for Tables

Guideline	Action
Always provide row or column headers in tables	The ADF Faces table based components provide proper HTML markup for row or column header data. Run the audit report. The <code>Verify that table columns have headers</code> audit rule warns when column header data is missing. Applications which use <code>trh:tableLayout</code> to construct data or layout tables are responsible for ensuring that such tables adhere to all Oracle accessibility guidelines.

Table 37-10 (Cont.) Style Guidelines for Tables

Guideline	Action
Provide a description for each table component using the <code>summary</code> attribute or <code>CAPTION</code> element.	Run the audit report. The <code>Verify that tables has summaries</code> audit rule warns when data tables are missing the <code>summary</code> attribute.
Ensure that layout tables do not use the <code>TH</code> element.	No action required. ADF Faces ensures that layout components do not use <code>TH</code> for layout tables.
Ensure that layout tables specify <code>summary=""</code> and do not have the <code>CAPTION</code> element	No action required. ADF Faces ensures that the layout components generate an empty summary for layout tables.
Provide correct reading sequence in a layout table	No action required. ADF Faces ensures that the reading sequence is correct for any layout tables that it generates.

How to Use WAI-ARIA Landmark Regions

The WAI-ARIA standard defines different sections of the page as different landmark regions. Together with WAI-ARIA roles, they convey information about the high-level structure of the page and facilitate navigation across landmark areas. This is particularly useful to users of assistive technologies such as screen readers.

ADF Faces includes landmark attributes for several layout components, as listed in [Table 37-11](#).

Table 37-11 ADF Faces Components with Landmark Attributes

Component	Attribute
decorativeBox	topLandmark
	centerLandmark
panelGroupLayout	landmark
panelSplitter	firstLandmark
	secondLandmark
panelStretchLayout	topLandmark
	startLandmark
	centerLandmark
	endLandmark
	bottomLandmark

These attributes can be set to one of the WAI-ARIA landmark roles, including:

- banner
- complimentary
- contentinfo

- main
- navigation
- search

When any of the landmark-related attributes is set, ADF Faces renders a role attribute with the value you specified.

Creating Accessible Active Data Components

The ADF Faces accessibility implementation generates standard W3C-defined aria attributes in the page's DOM (Document Object Model) of ADS (active data service) components. These attributes ensure updates to ADS components comply with accessibility guidelines so that a screen reader may announce meaningful output. You may customize what the screen reader may announce using pass-through attributes in the page.

Note:

All ADS components generate aria attributes in the page's DOM by default and therefore support the screen reader without requiring a special reader mode.

ADF has the following implementation of these standard aria attributes for interpreting change events on dynamically changing ADS components:

- `aria-live` indicates whether the screen reader should announce scalar value changes to ADS components. By default, ADF will generate an `aria-live` attribute with the value of `polite` for any ADS component that displays scalar values.
- `aria-atomic` is an optional attribute of both scalar and collection-based components (such as tables) that enables the screen reader to get additional context around the updated data, such as the label of the ADS component, when announcing new value changes.
- `aria-relevant` is an optional attribute of collection-based components (such as tables) that controls which type of updates to a collection are announced. By default, additions and updates are announced. With pass-through attributes this attribute can be changed to announce any combination of additions, updates and deletions.

Note:

For more details about these ADF-supported aria attributes and their allowed values, see https://www.w3.org/TR/wai-aria/states_and_properties.

ADF also supports the JSF specification to add custom attributes to the DOM, when you want to customize the response of the screen reader to ADS component updates. This capability is called `passthrough`, and is utilized to pass the aria attributes to the generated page markup. The ADF implementation of pass-through attributes supports

passing the attributes with the addition of the following namespaces as defined by `jcp.org` in the JSF or JSP page definition:

```
xmlns:f="http://xmlns.jcp.org/jsf/core" and xmlns:p="http://  
xmlns.jcp.org/jsf/passthrough"
```

For more information about ADF support for pass-through attributes, see [What You May Need to Know About Pass-Through Attributes](#).

Use Case Description

Two types of ADS components exist with regards to accessibility and screen reader responses to change events: scalar components and collection-based components. The accessibility implementation for scalar components, such as `activeOutputText`, relies on the screen reader default behavior and is controlled by standard aria pass-through attributes. Whereas, in the case of collection-based components including `af:table`, `af:tree`, and `af:treeTable`, ADF relies on the `AdfPage.PAGE.announceToAssistiveTechnology` API to craft a string for announcement and places the string in a separate section of the rendered page. The collection-based components do not use aria pass-through attributes to achieve the default response.

Default Screen Reader Responses

ADF supports the following default behavior when responding to change events in ADS components. Configuration by the user is not necessary to achieve the default screen reader response.

- For scalar components, such as `activeOutputText`, announce a new value when it is convenient so that interruptions of the screen reader are avoided. Only the changed value is announced (the component label is not included in the announcement) and the change will only be announced at the end of the current sentence, or after the user pauses typing. At runtime, ADF adds `aria-live="polite"` on the generated DOM of the component.
- For collection-based components, such as `tables`, announce row update with row details, announce row insert with row header only, and ignore row deletes. At runtime, the `AdfPage.PAGE.announceToAssistiveTechnology` API crafts a string for the announcement and places it in separate section of the generated page.

Modified Screen Reader Responses

In addition to the default screen reader response to active-data change events, the user may modify certain responses by configuring pass-through attributes in the page markup.

The following table describes the response modifications supported on scalar ADS components.

Table 37-12 Supported Modifications to Default Screen Reader Response for Scalar Components With Active Data

Custom Value Change Event Response	User Configuration
announce label and value change	User adds <code>aria-atomic="true"</code> on the containing <code>af:panelGroupLayout</code> of the component. At runtime, ADF passes through <code>aria-atomic="true"</code> and adds <code>aria-live="polite"</code> on the generated DOM of the containing the component.
allow interruptions to announce value change	User adds <code>aria-live="assertive"</code> on the containing <code>af:panelGroupLayout</code> of the component. At runtime, ADF passes through <code>aria-live="assertive"</code> on the generated DOM of the component.
allow interruptions to announce label and value change	User adds <code>aria-atomic="true"</code> and <code>aria-live="assertive"</code> on the containing <code>af:panelGroupLayout</code> of the component. At runtime, ADF passes through <code>aria-atomic="true"</code> and <code>aria-live="assertive"</code> on the generated DOM of the component.
turn off announcements	User adds <code>aria-live="off"</code> on the containing <code>af:panelGroupLayout</code> of the component. ADF passes through <code>aria-atomic="off"</code> on the DOM of the container.

The following table describes the response modifications supported on collection-based active-data components.

Table 37-13 Supported Modifications to Default Screen Reader Response for Collection-Based Components with Active Data

Row-Level Change Event Response	User Configuration
announce row insertions with row details (not only the row header)	User adds <code>data-aria-read-whole-row="true"</code> on the component. At runtime, ADF passes through <code>data-aria-read-whole-row="true"</code> on the DOM of the component.
announce row deletions with row header only	User adds <code>aria-relevant="all"</code> on the component. At runtime, ADF passes through <code>aria-relevant="all"</code> on the DOM of the component.
announce row deletions with row details (not only the row header)	User adds <code>aria-relevant="all"</code> and <code>data-aria-read-whole-row="true"</code> on the component. At runtime, ADF passes through <code>aria-relevant="all"</code> and <code>data-aria-read-whole-row="true"</code> on the DOM of the component.
revert to default behavior of screen reader software	User adds <code>aria-live="polite" "assertive"</code> on the component. At runtime, ADF passes through <code>aria-live="xx"</code> on the DOM of the component.
read the entire table on any updates	User adds <code>aria-live="polite"</code> and <code>aria-atomic="true"</code> on the component. At runtime, ADF passes through <code>aria-live="polite"</code> and <code>aria-atomic="true"</code> on the DOM of the component.

Table 37-13 (Cont.) Supported Modifications to Default Screen Reader Response for Collection-Based Components with Active Data

Row-Level Change Event Response	User Configuration
turn off announcements	User adds <code>aria-live="off"</code> on the component. At runtime, ADF passes through <code>aria-live="off"</code> on the DOM of the component.

How to Customize the Screen Reader Response for Scalar Active Data Components

By default, screen readers will announce a new value upon active data components updates. You can customize the default screen reader response for active data updates to achieve a more meaningful announcement of the component label and the new value.

To alter the default screen reader response for a scalar active data component:

1. In the source view of the page, locate the scalar active data component whose response you want to customize.
2. In the active data component or its containing panel, add the desired aria-prefixed markup attribute and value.

Tip: When more than one active data scalar component is contained in a panel, you can add the aria attribute to the containing panel. This will modify the screen reader response for all active data components based on the same aria attribute. In the following example, the `p:aria-atomic` attribute placed on the containing `af:panelGroupLayout` modifies the default screen reader response for the `af:outputText` component.

Note:

Any aria attribute that you use with scalar active data components must be designated by the "p:" namespace. For example, the `p:aria-atomic="true"` attribute identifies the namespace by the prepended `p:`. Additionally, the page must include these namespaces that enable the use of pass-through attributes: `xmlns:f="http://xmlns.jcp.org/jsf/core"` and `xmlns:p="http://xmlns.jcp.org/jsf/passthrough"`. For more information, see [What You May Need to Know About Pass-Through Attributes](#).

```
<f:view xmlns:f="http://xmlns.jcp.org/jsf/core" xmlns:h="http://java.sun.com/jsf/html" xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
  xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
  <af:document title="simple active text 2">
    <af:messages/>
    <af:form>
      <af:outputText value="Aria atomic set on the info to read label and
value"
                    inlineStyle="font:normal bolder 14pt Arial"/>
```

```
<af:spacer height="20"/>
<af:panelGroupLayout id="plm1" p:aria-atomic="true">
  <af:outputText value="Active Output Text: "/>
  <af:activeOutputText id="text"
value="#{bindings.beanData.collectionModel.firstRow.name}"/>
</af:panelGroupLayout>
<af:spacer height="20"/>
</af:form>
</af:document>
</f:view>
```

When you use `p:aria-atomic="true"`, the screen reader will announce the label together with the new value on the active data component without interrupting the user's current task. Other customizations that you can specify allow the announcements to interrupt the current task so the user will be immediately notified of a change:

- `p:aria-live="assertive"` to allow interruptions to announce only value changes.
- `p:aria-atomic="true"` and `p:aria-live="assertive"` to allow interruptions to announce label and value change together

When you want to turn off announcements for specific active data components, add `p:aria-live="off"` to the component or components' containing panel.

How to Customize the Screen Reader Response for Collection-Based Active Data Components

Collection-based active data components including `af:table`, `af:tree`, and `af:treeTable` provide default accessibility behavior to ensure that a screen reader will announce a row/node update, row/node insertion, and row/node deletion. You can customize the default screen reader response for collection-based active data updates to achieve a more meaningful announcement of the update in the context of the collection.

Before you begin:

It may be helpful to have an understanding of the accessibility use cases for collection-based components. For more information, see [Creating Accessible Active Data Components](#).

It may be helpful to have an understanding of the default screen response to active data updates for collection-based components. For more information, see [Creating Accessible Active Data Components](#).

You must ensure that your collection-based component adheres to the guidelines described in [ADF Faces Component Accessibility Guidelines](#). For example, a table, tree, or treeTable component must have a short description defined by the summary property and a table component must have a row header that resolves to readable text.

To alter the default screen reader response for a collection-based active data component:

1. In the source view of the page, locate the collection-based active data component whose response you want to customize.
2. In the active data component, add the desired aria-prefixed markup attribute and value.

Collection-based components rely on a variety of aria attributes placed on the component itself to modify the default screen reader response to active data updates.



Note:

Any aria attribute that you use with collection-based active data components must be designated by the “p:” namespace. For example, the `p:aria-relevant="all"` attribute identifies the namespace by the prepended `p:`. Additionally, the page must include these namespaces that enable the use of pass-through attributes: `xmlns:f="http://xmlns.jcp.org/jsf/core"` and `xmlns:p="http://xmlns.jcp.org/jsf/passthrough"`. For more information, see [What You May Need to Know About Pass-Through Attributes](#).

The following table shows the attributes that you add to the `af:table` component to achieve the desired customization. In this example, the table identifies updates to employee address records, where the table is named `address` and the allowed customization are described for a row identifying employee `Mark`.

Table 37-14 Allowed Accessibility Customizations for Table Components with Active Data

Customize Response	User Configuration	Announcement Format	Announcement Example
announce row insertions with row details (not only the row header)	<code>p:data-aria-read-whole-row="true"</code>	Row inserted in {table's name} {textual row content}	Row inserted in address table Mark Birch Lane 95050 Redwood Shores
announce row deletions with row header only	<code>p:aria-relevant="all"</code>	Row deleted from {table's name} {rowheader}	Row deleted from address table Mark
announce row deletions with row details (not only the row header)	<code>p:aria-relevant="all"</code> and <code>p:data-aria-read-whole-row="true"</code>	Row deleted from {table's name} {textual row content}	Row deleted from address table Mark Birch Lane 95050 Mountain View
read the entire table upon any updates	<code>p:aria-live="polite"</code> and <code>p:aria-atomic="true"</code>	Reads the entire table	Not provided

The following table shows the attributes that you add to the `af:treeTable` component to achieve the desired customization. In this example, the table refers to employee address records, where the tree table is named `address` and the

allowed customization are described for a non-root node identifying employee Mark, which is a child in the tree table of parent node Twain.

Table 37-15 Allowed Accessibility Customizations for Tree Table Components with Active Data

Customize Response	User Configuration	Announcement Format	Announcement Example
announce row insertions made on the tree table root node with row details (not only the row header)	<code>p:data-aria-read-whole-row="true"</code>	Row inserted in {tree table's name} {textual row content}	Row inserted in address tree table Mark Birch Lane 95050 Redwood Shores
announce row deletions made on the tree table non-root node with rows node stamp	<code>p:aria-relevant="all"</code>	Row deleted from {tree table's name} parent node {immediate parent row's node text} {row's node stamp text}	Row deleted from address tree table, parent node Twain, Mark
announce row deletions made on tree table non-root node with row context	<code>p:aria-relevant="all" and data-aria-read-whole-row="true"</code>	Row deleted from {tree table's name} parent node {immediate parent row's node text} {textual row content}	Row deleted from address tree table, parent node Twain, Mark Birch Lane 95050 Mountain View
read the entire tree table upon any updates	<code>p:aria-live="polite" and p:aria-atomic="true"</code>	Reads the entire tree table	Not provided

The following table shows the attributes that you add to the `af:tree` component to achieve the desired customization. In this example, the table refers to employee address records, where the tree is named `address` and the allowed customizations are described for a node identifying employee Mark, which is a child in the tree of parent node Twain.

Table 37-16 Allowed Accessibility Customizations for Tree Components with Active Data

Customize Response	User Configuration	Announcement Format	Announcement Example
announce row deletions made on the tree root node	p:aria-relevant="all"	Node deleted from {tree's name} {node text}	Node deleted from address tree, Mark
announce row deletions made on tree non-root node	p:aria-relevant="all"	Node deleted from {tree's name} parent node {immediate parent's node text} {node text}	Node deleted from address tree, parent node Twain, Mark
read the entire tree upon any updates	p:aria-live="polite" and p:aria-atomic="true"	Reads the entire tree table	Not provided

Alternatively, when you can prefer to disable the announcements supported by the ADF `AdfPage.PAGE.announceToAssistiveTechnology` API for collection-based active data components, you have two options:

- Turn off ADF-supported announcements, but allow announcements supported by the default behavior of the screen reader and browser, then set `aria-live="polite" | "assertive"` on the collection-based component.
- Turn off announcements entirely, then set `aria-live="off"` on the collection-based component.

What You May Need to Know About Pass-Through Attributes

The ADF implementation of pass-through attributes supports passing the attributes through to both JSF and JSP markup. JSF web pages that you create with ADF components also support all JSF specification formats for implementing pass-through attributes. However, in the case of JSP pages, your page must use the ADF tag `af:passThroughAttribute` to add the custom attributes on the runtime generated DOM (Document Object Model) for the page.

As a prerequisite, JSF and JSP page definitions must specify the following namespaces as defined by `jcp.org`:

- `xmlns:f="http://xmlns.jcp.org/jsf/core"`
- `xmlns:p="http://xmlns.jcp.org/jsf/passthrough"`

For example:

```
<f:view xmlns:f="http://xmlns.jcp.org/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
        xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
```

 **Note:**

Please note that the `xmlns:f="http://xmlns.jcp.org/jsf/core"` namespace is the replacement for `xmlns:f="http://java.sun.com/jsf/core"`. Pass through only works with the new version.

You may work with pass-through attributes in various ways:

- Using the `p:name="value"` tag
- Using the `f:passThroughAttribute name="xx" value="yy"` tag
- Using the `f:passThroughAttributes value="#{binding.mapofpassthroughattributes}"` with an EL-bound pass-through value, where the value must evaluate to a `Map<String, Object>` having all the pass-through name- value combinations you want to add to the runtime generated markup.

The following JSF snippet shows examples of all three ways to use pass-through attributes in a page.

```
<f:view xmlns:f="http://xmlns.jcp.org/jsf/core" xmlns:h="http://java.sun.com/jsf/html"
xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <h:head>
      <title>untitled1</title>
    </h:head>
    <h:body>
      <h:form>
        <af:inputText columns="150" id="t1" value="this is a simple one with a
passthrough"
          p:type="email">
        </af:inputText>
        <af:spacer></af:spacer>
        <af:inputText columns="150" id="t2" value="this one uses
f:passThroughAttribute">
          <f:passThroughAttribute name="type" value="email"/>
        </af:inputText>
        <af:spacer></af:spacer>
        <af:inputText columns="150" id="t5" value="uses an
f:passThroughAttributes map">
          <f:passThroughAttributes value="#{SampleBean.passthroughs}"/>
        </af:inputText>
        <af:spacer></af:spacer>
      </h:form>
    </h:body>
  </html>
</f:view>
```

The following runtime generated DOM shows the pass-through result for each of the above usages, where the `type` attribute of the table element is `email`. Only the table element is stamped for the pass-through attribute because it is the outermost element and none of the inner elements have the pass-through attribute.

 **Note:**

At runtime, pass-through attributes are stamped on the *outermost* markup element of the component with the pass-through attribute definition.

```
<table cellpadding="0" cellspacing="0" border="0" role="presentation" summary=""
id="j_idt5:t1" class="af_inputText"
  type="email">
  <tbody>
    <tr>
      <td class="af_inputText_label"></td>
      <td valign="top" nowrap="" class="AFContentCell">
        <input type="text" value="this is a simple one with a passthrough"
size="150" class="af_inputText_content"
          style="width:auto" name="j_idt5:t1" id="j_idt5:t1::content">
      </td>
    </tr>
  </tbody>
</table>
```

```
<table cellpadding="0" cellspacing="0" border="0" role="presentation" summary=""
id="j_idt5:t2" class="af_inputText"
  type="email">
  <tbody>
    <tr>
      <td class="af_inputText_label"></td>
      <td valign="top" nowrap="" class="AFContentCell">
        <input type="text" value="this one uses f:passThroughAttribute"
size="150" class="af_inputText_content"
          style="width:auto" name="j_idt5:t2" id="j_idt5:t2::content">
      </td>
    </tr>
  </tbody>
</table>
```

```
<table cellpadding="0" cellspacing="0" border="0" role="presentation" summary=""
id="j_idt5:t5" class="af_inputText"
  type="email" genre="personal">
  <tbody>
    <tr>
      <td class="af_inputText_label"></td>
      <td valign="top" nowrap="" class="AFContentCell">
        <input type="text" value="uses an f:passThroughAttributes map"
size="150" class="af_inputText_content"
          style="width:auto" name="j_idt5:t5" id="j_idt5:t5::content">
      </td>
    </tr>
  </tbody>
</table>
```

When you use the ADF tag in either the JSF or JSP page, observe the following conditions and limitations:

1. `af:passThroughAttribute` value can be expression language (EL) bound where the `name` property is a string.
2. `af:passThroughAttribute` can be used to overwrite any rendered default value.

3. `af:passThroughAttribute` is stamped only on the root element (outermost) of the component. None of the nested inner elements should have them.
4. `af:passThroughAttribute` with `name` property defined as `elementName` is treated specially. If an `af:passThroughAttribute` is set with `elementName`, then the element's name itself changes to its value. For example:

```
<af:passThroughAttribute name="elementName" value="test" ...>
```

will render the root element of the component in the generated page markup as:

```
<test>.....</test>
```

Running Accessibility Audit Rules

JDeveloper supports generating accessibility compliance reports for ADF applications by allowing you to run accessibility-specific audit rules on the application.

JDeveloper provides ADF Faces accessibility audit rules to investigate and report compliance with many of the common requirements described in [ADF Faces Component Accessibility Guidelines](#).

How to Create an Audit Profile

You can create an audit profile from the Preferences dialog.

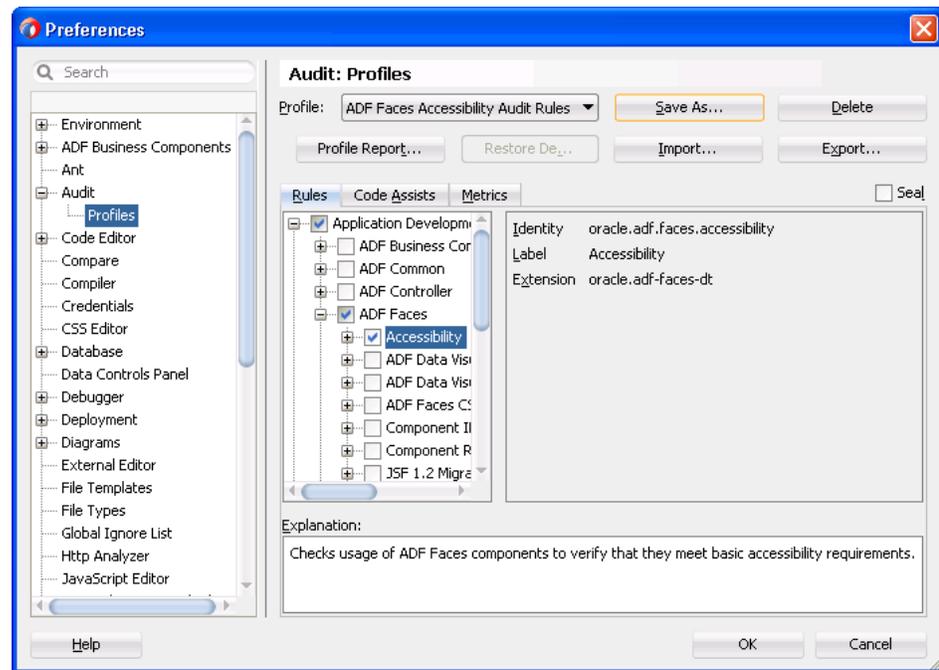
Before you begin:

It may be helpful to have an understanding of accessibility audit rules. For more information, see [Running Accessibility Audit Rules](#). You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Information for Accessibility Support in ADF Pages](#).

To create an audit profile:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, choose **Audit > Profiles**.
3. In the Audit: Profiles dialog, clear all checkboxes, and then select the **Application Development Framework > ADF Faces > Accessibility** checkbox.
4. Click **Save As** and save the profile with a unique name.

[Figure 37-1](#) illustrates the settings of the Audit: Profiles dialog to create an accessibility audit profile.

Figure 37-1 Audit Profile Settings for ADF Faces Accessibility

5. Click **OK**.

How to Run Audit Report

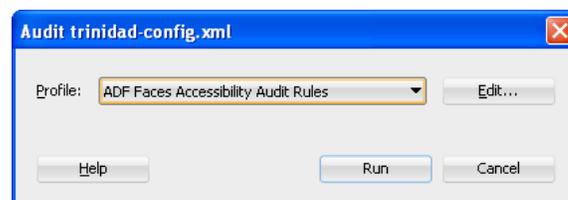
Running an audit report requires creating and running an audit profile.

Before you begin:

It may be helpful to have an understanding of accessibility audit rules. For more information, see [Running Accessibility Audit Rules](#). You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Information for Accessibility Support in ADF Pages](#).

To run the audit report:

1. From the main menu, choose **Build > Audit target**.
2. In the Audit dialog, from the Profile dropdown menu, choose the ADF Faces accessibility audit profile you created.

Figure 37-2 Audit Dialog To Run an Audit Report

3. Click **Run** to generate the report.

The audit report results are displayed in the Log window. After the report generation is complete, you can export the report to an HTML file by clicking the **Export** icon in the Log window toolbar.

Allowing User Customization on JSF Pages

This chapter describes how changes to certain UI components that the user makes at runtime can persist for the duration of the session.

Alternatively, you can configure your application so that changes persist in a permanent data repository. Doing so means that the changes remain whenever the user reenters the application. To allow this permanent persistence, you need to use the Oracle Metadata Service (MDS), which is part of the full Fusion technology stack. Using MDS and the full Fusion stack also provides the following additional persistence functionality:

- Persisting additional attribute values
- Persisting search criteria
- Persisting the results of drag and drop gestures in the UI
- Reordering components on a page at runtime
- Adding and removing components and facets from the page at runtime

For information and procedures for using Oracle MDS, see the "Allowing User Customizations at Runtime" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

- [About User Customization](#)
- [Implementing Session Change Persistence](#)

About User Customization

Users of ADF applications can make personalization changes to the ADF Faces components that, depending on the application configuration, may or may not persist for the duration of the user session. However, implicit personalization is supported by ADF Faces for a wide variety of UI aspects.

Many ADF Faces components allow users to change the display of the component at runtime. For example, a user can change the location of the splitter in the `panelSplitter` component or change whether or not a panel displays detail contents. By default, these changes live only as long as the page request. If the user leaves the page and then returns, the component displays in the manner it is configured by default. However, you can configure your application so that the changes persist through the length of the user's session. This way the changes will stay in place until the user leaves the application.

[Table 38-1](#) shows the changes by component that provide default personalization capabilities:

Table 38-1 Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
panelBox showDetail showDetailHeader showDetailItem	disclosed	Display or hide content	Users can display or hide content using an icon in the header. Detail content will either display or be hidden, based on the last action of the user.
showDetailItem (used in a panelAccordion component)	flex	Height of multiple components	The heights of multiple showDetailItem components are determined by their relative value of the flex attribute. The showDetailItem components with larger flex values will be taller than those with smaller values. Users can change these proportions, and the new values will be persisted.
showDetailItem (used in a panelAccordion component)	inflexibleHeight	Size of a panel	Users can change the size of a panel, and that size will remain.
panelSplitter	collapsed	Collapse splitter sides	Users can collapse either side of the splitter. The collapsed state will remain as last configured by the user.
panelSplitter	splitterPosition	Splitter position	The position of the splitter in the panel will remain where last moved by user.
richTextEditor	editMode	Selected edit mode	The editor will display using the mode (either WYSIWYG or source) last selected by the user.
calendar	activeDay	Active day	The day considered active in the current display will remain the active day.
calendar	view	Display activities view	The view (day, week, month, or list) that currently displays activities will be retained.
panelWindow dialog	contentHeight	Height of a panel window or a dialog	Users can change the height of a panelWindow or dialog popup component, and that height will remain.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
panelWindow dialog	contentWidth	Width of a panel window or a dialog	Users can change the width of a panelWindow or dialog popup component, and that width will remain.
button link	windowHeight	Inline dialog height	When an inline popup dialog is launched using the ADF Faces dialog framework or an ADF taskflow, if the user manually resizes the dialog, any associated windowHeight value on the command component that launched the dialog is also changed and will remain. This feature only applies to inline dialogs and not browser window dialogs.
button link	windowWidth	Inline dialog width	When an inline popup dialog is launched using the ADF Faces dialog framework or an ADF taskflow, if the user manually resizes the dialog, any associated windowWidth value on the command component that launched the dialog is also changed and will remain. This feature only applies to inline dialogs and not browser window dialogs.
column	displayIndex	Column reordering	ADF Faces columns can be reordered by the user at runtime. The displayIndex attribute determines the order of the columns. (By default, the value is set to -1 for each column, which means the columns will display in the same order as the data source). When a user moves a column, the value on each column is changed to reflect the new order. These new values will be persisted.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
column	frozen	Column scrollability	ADF Faces columns can be frozen so that they will not scroll. When a column's <code>frozen</code> attribute is set to <code>true</code> , all columns before that column (based on the <code>displayIndex</code> value) will not scroll. When you use the table with a <code>panelCollection</code> component, you can configure the table so that a button appears that allows the user to freeze a column. For more information, see How to Display a Table on a Page .
column	<code>noWrap</code>	Column text set to wrap or not	The content of the column will either wrap or not. You need to create code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from <code>true</code> to <code>false</code> .
column	<code>selected</code>	Selected column	The selected column is based on the column last selected by the user.
column	<code>visible</code>	Column set to be visible or not	The column will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from <code>true</code> to <code>false</code> .
column	<code>width</code>	Column width	The width of the column will remain the same size as the user last set it.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
table	filterVisible	Display or hide filter	ADF Faces tables can contain a component that allows users to filter the table rows by an attribute value. For a table that is configured to use a filter, the filter will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a button that allows a user to toggle the value from true to false.
dvt:areaGraph dvt:barGraph dvt:bubbleGraph dvt:comboGraph dvt:horizontalBarGraph dvt:lineGraph dvt:scatterGraph	timeRangeMode	Mode of time range to display data on graph time axis	The time range for the data displayed on a graph time axis can be specified for all data visualization graph components. By default, all data is displayed. The time range can also be set for a relative time range from the last or first data point, or an explicit time range. You will need to write code that allows the user to change this attribute value. For example, you might create a dropdown list to choose the time range for a graph.
dvt:ganttLegend	visible	Display or hide legend for Gantt chart	The legend for data visualization project, resource utilization, and scheduling Gantt chart components will either be visible or not inside the information panel. You will need to write code that allows the user to change this attribute value, for example, a hide and show button to display the legend.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
dvt:hierarchyViewer	layout	Hierarchy viewer layout options	The data visualization hierarchy viewer component supports nine hierarchy layout options including a top-to-bottom vertical, tree, circle, radial, and so on. Users can change the layout in the map control panel and the last selected layout will be retained.
dvt:map	mapZoom	Map zoom level	This data visualization geographic map component attribute specifies the beginning zoom level of the map. The zoom levels are defined in the map cache instance as part of the base map. You will need to write code that allows the user to change this attribute value.
dvt:map	srid	Map spatial reference id	This data visualization geographic map component attribute specifies the srid (spatial reference id) of all the coordinates of the map, which includes the center of the map, defined by starting X and starting Y, and all the points in the point theme. You will need to write code that allows the user to change this attribute value.
dvt:map	startingX, startingY	Map X and Y coordinates	This data visualization geographic map component attribute specifies the X and Y coordinate of the center of the map. The srid for the coordinate is specified in the srid attribute. If the srid attribute is not specified, this attribute assumes that its value is the longitude of the center of the map. You will need to write code that allows the user to change this attribute value.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
dvt:projectGantt dvt:resource UtilizationGantt dvt:schedulingGantt	splitterPosition	Gantt chart splitter position	The position of the splitter in the panel will remain where last moved by user.
dvt:timeAxis	scale	Gantt chart time axes	Data visualization components for project, resource utilization, and scheduling Gantt charts use this facet to specify the major and minor time axes in the Gantt chart. The time scale (twoyears, year, halfyears, quarters, twomonths, months, weeks, twoweeks, days, sixhours, threehours, hours, halfhours, quarterhours) can be set by the user using the menu bar View menu and the selection will be retained. Note that a custom time scale can also be named for this component value.
dvt:timeSelector	explicitStart, explicitEnd	Graph start and end dates	Data visualization area, bar, combo, line, scatter, and bubble graph components use this child tag attribute to specify the explicit start and end dates for the time selector. Only value-binding is supported for this attribute. You will need to write code that allows the user to change this attribute value.

Table 38-1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	UI Aspect	Affect at Runtime
dvt:treeMap	layout	Treemap hierarchy layout options	The data visualization treemap component supports three hierarchy layout options including squarified (nodes laid out as square as possible) and slice and dice horizontal or vertical (nodes are laid out horizontally or vertically first across the width of the treemap and then vertically or horizontally across the height of the treemap). You will need to write code that allows the user to change the layout.

User Customization Use Cases and Examples

You can configure an application so that the value of the attributes listed in [Table 38-1](#) can be persisted through the length of the user's session. For example, say your application contains a table, and a user adjusts the width of a column so that the contents all display on one line. If you configure your application to use session change persistence, when the user leaves and then returns to that page, the column will still be expanded to the previously set width.

Note:

For additional functionality, you can configure your application so that changes persist in a permanent data repository, meaning they will persist for that user across multiple sessions. To allow this permanent persistence, you need to use the full Fusion technology stack. For more information, see the "Allowing User Customizations at Runtime" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Implementing Session Change Persistence

Persistence of user interface customizations made by users of ADF applications is controlled by a `web.xml` configuration setting. When the setting is enabled, ADF Faces persists user interface changes for the duration of the user's session.

In order for the application to persist user changes to the session, you must configure your project to enable customizations.

How to Implement Session Change Persistence

You configure your application to enable customizations in the `web.xml` file.

To implement session change persistence:

1. In the Applications window, double-click the web project.
2. In the Project Properties dialog, select the **ADF View** node.
3. On the ADF View page, activate the **Enable User Customizations** checkbox, select the **For Duration of Session** radio button, and click **OK**.

What Happens When You Configure Your Application to Use Change Persistence

When you elect to save changes to the session, JDeveloper adds the `CHANGE_PERSISTENCE` context parameter to the `web.xml` file, and sets the value to `session`. This context parameter registers the `ChangeManager` class that will be used to handle persistence. The following example shows the context parameter in the `web.xml` file.

```
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHANGE_PERSISTENCE</param-name>
  <param-value>session</param-value>
</context-param>
```

What Happens at Runtime: How Changes are Persisted

When an application is configured to persist changes to the session, any changes are recorded in a session variable in a data structure that is indexed according to the view ID. Before the `RENDER_RESPONSE` JSF phase begins, the tag action classes look up all changes for a given component and apply the changes in the same order as they were added. This means that the changes registered through the session will be applied only during subsequent requests in the same session.

What You May Need to Know About Using Change Persistence on Templates and Regions

When you use session persistence, changes are recorded and restored on components against the `viewId` for the given session. As a result, when the change is applied on a component that belongs to a fragment or page template, it is applicable only in scope of the page that uses the fragment or template. It does not span all pages that consume the fragment or template. For example, say your project has the `pageOne.jsf` and `pageTwo.jsf` JSF pages, and they both contain the fragment defined in the `region.jsff` page fragment, which in turn contains a `showDetail` component. When the `pageOne.jsf` JSF page is rendered and the `disclosed` attribute on the `showDetail` component changes, the implicit attribute change is recorded and will be applied only for the `pageOne.jsf` page. If the user navigates to the `pageTwo.jsf` page, no attribute change is applied.

Adding Drag and Drop Functionality

This chapter describes how to add drag and drop functionality to your pages, which allows users to drag the values of attributes or objects from one component to another, or allows users to drag and drop components. It describes how to add drag and drop functionality for attributes, objects, collections, components, calendars, and supported DVT components. It also describes how to drag and drop functionality into and out of a `panelDashboard` component.

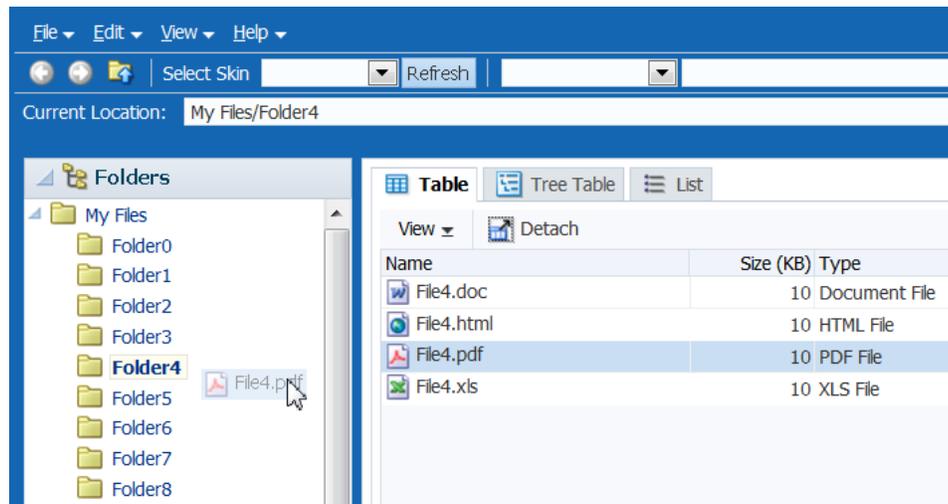
This chapter includes the following sections:

- [About Drag and Drop Functionality](#)
- [Adding Drag and Drop Functionality for Attributes](#)
- [Adding Drag and Drop Functionality for Objects](#)
- [Adding Drag and Drop Functionality for Collections](#)
- [Adding Drag and Drop Functionality for Components](#)
- [Adding Drag and Drop Functionality Into and Out of a `panelDashboard` Component](#)
- [Adding Drag and Drop Functionality to a Calendar](#)
- [Adding Drag and Drop Functionality for DVT Components](#)

About Drag and Drop Functionality

Drag and drop functionality that you implement for ADF Faces components supports the end-user experience of interactively dragging a UI element, such as an attribute value or a UI component, and dropping it onto a supported target destination in the user interface of the displayed web page. Users of ADF applications can use ADF Faces drag and drop functionality that has been configured to work with attributes, objects, collections, components, calendars, and various DVT components.

The ADF Faces framework provides the ability to drag and drop items from one place to another on a page. In most cases, drag and drop can easily be implemented by adding the appropriate tags to the source and target and implementing code in a managed bean. Drag and drop provides users with the GUI experience that is expected in web applications. For example, in the File Explorer application, you can drag a file from the **Table** tab and drop it into another directory folder, as shown in [Figure 39-1](#).

Figure 39-1 Drag and Drop in the File Explorer Application

In this scenario, you are actually dragging an object from one collection (Folder0) and dropping it into another collection (Folder2). This is one of the many supported drag and drop scenarios. ADF Faces supports the following scenarios:

- Dragging an attribute value from one component instance and copying it to another. For example, a user might be able to drag an `outputText` component onto an `inputText` component, which would result in the `value` attribute of the `outputText` component becoming the `value` attribute on the `inputText` component.
- Dragging the value of one object and dropping it so that it becomes the value of another object. For example, a user might be able to drag an `outputText` component onto another `outputText` component, which would result in an array of `String` objects populating the `text` attribute of the second `outputText` component.
- Dragging an object from one collection and dropping it into another, as shown in [Figure 39-1](#).
- Dragging a component from one place on a page to another. For example, a user might be able to drag an existing `panelBox` component to a new place within a `panelGrid` component.
- Dragging an activity in a calendar from one start time or date to another.
- Dragging a component into or out of a `panelDashboard` component.
- Dropping an object from another component into a DVT Pareto or stock graph.
- Dragging an object from a DVT Gantt chart to another component.
- Dragging a node from or dropping an object to DVT treemap and sunburst components.
- Dragging and dropping one or more nodes within DVT hierarchy viewers, dragging one or more nodes from a hierarchy viewer to another component, or dragging from one or more components to a hierarchy viewer.
- Dragging and dropping an event from a DVT timeline to a collection component such as a table, or dragging and dropping a row from a table of time-based events into a timeline.

When users click on a source and begin to drag, the browser displays the element being dragged as a ghost element attached to the mouse pointer. Once the ghost element hovers over a valid target, the target component shows some feedback (for example, it becomes highlighted and the cursor changes to indicate the target is valid). If the user drags the ghost element over an invalid target, the cursor changes to indicate that the target is not valid.

When dragging attribute values, the user can only copy the value to the target. For all other drag and drop scenarios, on the drop, the element can be copied (copy and paste), moved (cut and paste), or linked (creating a shortcut for a file in a directory in which the link is a reference to the real file object).

The component that will be dragged and that contains the value is called the source. The component that will accept the drop is called the target. You use a specific tag as a child to the source and target components that tells the framework to allow the drop. [Table 39-1](#) shows the different drag and drop scenarios, the valid source(s) and target(s), and the associated tags to be used for that scenario.

Table 39-1 Drag and Drop Scenarios

Scenario	Source	Target
Dragging an attribute value	An attribute value on a component Tag: attributeDragSource	An attribute value on another component, as long as it is the same object type Tag: attributeDropTarget
Dragging an object from one component to another	Any component Tag: attributeDragSource	Any component Tag: dropTarget
Dragging an item from one collection and dropping it into another	table, tree, and treeTable components Tag: dragSource	table, tree, and treeTable components Tag: collectionDropTarget
Dragging a component from one container to another	Any component Tag: componentDragSource	Any component Tag: dropTarget
Dragging a calendar activity from one start time or date to another	calendarActivity object Tag: None needed	calendar component Tag: calendarDropTarget
Dragging a panelBox component into a panelDashboard component.	panelBox component Tag: componentDragSource	panelDashboard component Tag: dataFlavor
Dragging a panelBox component out of a panelDashboard component.	panelBox component in a panelDashboard component Tag: componentDragSource	Any component Tag: dropTarget

Table 39-1 (Cont.) Drag and Drop Scenarios

Scenario	Source	Target
Dropping an object from another component into a Pareto or stock graph.	Any component Tag: dragSource	paretoGraph or stockGraph component Tag: dropTarget
Dragging an object from a DVT Gantt chart and dropping it on another component	Gantt chart Tag: dragSource	Any component Tag: dropTarget
Dragging a node from a DVT hierarchy viewer, sunburst, or treemap and dropping it on another component	hierarchyViewer, sunburst, or treemap component Tag: dragSource	Any component Tag: dropTarget
Dragging an event from a timeline and dropping it into a collection component	timeline components Tag: dragSource	table, tree, and treeTable components Tag: collectionDropTarget

You can restrict the type of the object that can be dropped on a target by adding a `dataFlavor` tag. This helps when the target can accept only one object type, but the source may be one of a number of different types. The `dataFlavor` tag also allows you to set multiple types so that the target can accept objects from more than one source or from a source that may contain more than one type. For the drop to be successful, both the target and the source must contain the `dataFlavor` tag, and both the Java type that the `dataFlavor` encapsulates along with the discriminant need to be same between the source and the target.

 **Note:**

Drag and drop functionality is not supported between windows. Any drag that extends past the window boundaries will be canceled. Drag and drop functionality is supported between popup windows and the base page for the popup.

Also note that drag and drop functionality is not accessible; that is, there are no keyboard strokes that can be used to execute a drag and drop. Therefore, if your application requires all functionality to be accessible, you must provide this logic. For example, your page might also present users with a method for selecting objects and a Move button or menu item that allows them to move those selected objects.

Additional Functionality for Drag and Drop

You may find it helpful to understand other ADF Faces features before you implement drag and drop. Following are links to other sections that may be useful for implementing drag and drop.

- **Managed beans:** You may be using managed beans for your code. For information about using managed beans, see [Creating and Using Managed Beans](#).
- **Events:** Table and tree components fire both server-side and client-side events that you can have your application react to by executing some logic. For more information, see [Handling Events](#).

Adding Drag and Drop Functionality for Attributes

You can enable ADF Faces drag and drop functionality for attribute values of ADF Faces components. Drag and drop functionality that you implement for ADF Faces components supports the end-user experience of interactively dragging the attribute value of one component and dropping it onto a supported target component in the user interface of the displayed web page.

You add drag and drop functionality for attributes by defining one component's attribute to be a target and another component's attribute to be a source.

Note:

The target and source attribute values must both be the same data type. For example, attribute drag and drop is available when both the source and target are of type String. If they are both of type number, they both use the same converters.

How to add Drag and Drop Functionality

You can drag and drop your target and source components that are already on the JSF page.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality for Attributes](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

To add drag and drop functionality for attributes:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop an **Attribute Drop Target** as a child to the target component on the page.
2. In the Insert Attribute Drop Target dialog, use the **Attribute** dropdown to select the attribute that will be populated by the drag and drop action. This dropdown list shows all valid attributes on the target component.
3. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop an **Attribute Drag Source** as a child to the component that can provide a value for the target.

4. In the Insert Attribute Drag Source dialog, use the **Attribute** dropdown to select the attribute whose value will be used to populate the target attribute. This dropdown list shows all valid attributes on the source component.

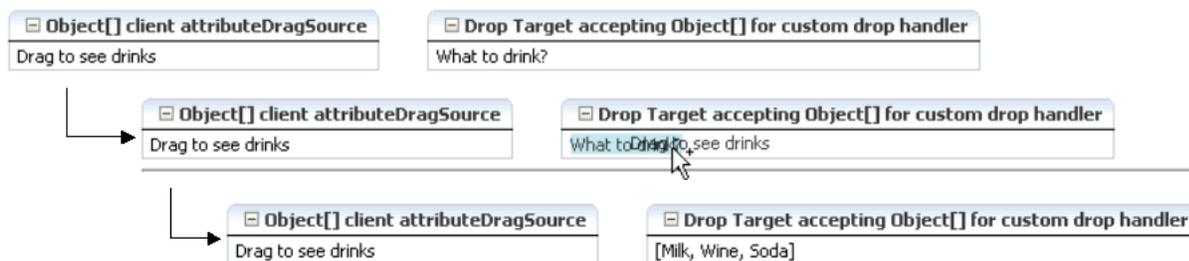
Adding Drag and Drop Functionality for Objects

You can enable ADF Faces drag and drop functionality for objects that are not attribute values displayed by ADF Faces components. Drag and drop functionality that you implement for ADF Faces components supports the end-user experience of interactively dragging the value of an object and dropping it onto a supported target object in the user interface of the displayed web page.

When you want users to be able to drag things other than attribute values, or you want users to be able to do something other than copy attributes from one component to another, you use the `dropTarget` tag. Additionally, use the `DataFlavor` object to determine the valid Java types of sources for the drop target. Because there may be several drop targets and drag sources, you can further restrict valid combinations by using discriminant values. You also must implement any required functionality in response to the drag and drop action.

For example, suppose you have an `outputText` component with an array of `Strings` and you want the user to be able to drag the `outputText` component to a `panelBox` component and have the `panelBox` display the `String` array, as shown in [Figure 39-2](#).

Figure 39-2 Dragging and Dropping an Array Object



The `outputText` component contains an `attributeDragSource` tag. However, because you want to drag an array of `String` values from the `outputText` component, you must use the `dropTarget` tag instead of the `attributeDropTarget` tag on the target `outputText` component. Also use a `dataFlavor` tag to ensure that only an array object will be accepted on the target.

You can also define a discriminant value for the `dataFlavor` tag. This is helpful if you have two targets and two sources, all with the same object type. By creating a discriminant value, you can be sure that each target will accept only valid sources. For example, suppose you have two targets that both accept an `EMPLOYEE` object, `TargetA` and `TargetB`. Suppose you also have two sources, both of which are `EMPLOYEE` objects. By setting a discriminant value on `TargetA` with a value of `alpha`, only the `EMPLOYEE` source that provides the discriminant value of `alpha` will be accepted.

You also must implement a listener for the drop event. The object of the drop event is called the `Transferable`, which contains the payload of the drop. Your listener must

access the `Transferable` object, and from there, use the `DataFlavor` object to verify that the object can be dropped. You then use the drop event to get the target component and update the property with the dropped object. More details about this listener are covered in the procedure in [Adding Drag and Drop Functionality for DVT Components](#).

How to Add Drag and Drop Functionality for a Single Object

To add drag and drop functionality, first add tags to a component that define it as a target for a drag and drop action. Then implement the event handler method that will handle the logic for the drag and drop action. Last, you define the sources for the drag and drop.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality for Objects](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

You will need to complete this task:

Create the source and target components on the page.

To add drag and drop functionality:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Drop Target** and drop it as a child to the target component on the page.
2. In the Insert Drop Target dialog, enter an expression that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 5).

For information about using managed beans, see [Creating and Using Managed Beans](#).

Tip:

You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see [What You May Need to Know About Using the ClientDropListener](#).

3. In the Insert Data Flavor dialog, enter the class for the object that can be dropped onto the target, for example `java.lang.Object`. This selection will be used to create a `dataFlavor` tag, which determines the type of object that can be dropped onto the target, for example a `String` or a `Date`. Multiple `dataFlavor` tags are allowed under a single drop target to allow the drop target to accept any of those types.

 **Tip:**

To specify a typed array in a `DataFlavor` tag, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

4. In the Structure window, select the `dropTarget` tag. In the Properties window, select a value for the `actions` attribute. This defines what actions are supported by the drop target. Valid values can be `COPY` (copy and paste), `MOVE` (cut and paste), and `LINK` (copy and paste as a link), for example:.

```
MOVE COPY
```

If no actions are specified, the default is `COPY`.

The following example shows the code for a `dropTarget` component inserted into an `panelBox` component that takes an array object as a drop target. Note that because an action was not defined, the only allowed action will be `COPY`.

```
<af:panelBox text="PanelBox2">
  <f:facet name="toolbar"/>
  <af:dropTarget dropListener="#{myBean.handleDrop}">
    <af:dataFlavor flavorClass="java.lang.Object[]" />
  </af:dropTarget>
</af:panelBox>
```

5. In the managed bean referenced in the EL expression created in Step 2, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

This method must take a `DropEvent` event as a parameter and return a `DnDAction` object, which is the action that will be performed when the source is dropped. Valid return values are `DnDAction.COPY`, `DnDAction.MOVE`, and `DnDAction.LINK`, and were set when you defined the target attribute in Step 4. This method should check the `DropEvent` event to determine whether or not it will accept the drop. If the method accepts the drop, it should perform the drop and return the `DnDAction` object it performed. Otherwise, it should return `DnDAction.NONE` to indicate that the drop was rejected.

The method must also check for the presence for each `dataFlavor` object in preference order.

 **Tip:**

If your target has more than one defined `dataFlavor` object, then you can use the `Transferable.getSuitableTransferData()` method, which returns a `List` of `TransferData` objects available in the `Transferable` object in order, from highest suitability to lowest.

The `DataFlavor` object defines the type of data being dropped, for example `java.lang.Object`, and must be as defined in the `DataFlavor` tag on the JSP, as created in Step 3.

 **Tip:**

To specify a typed array in a `DataFlavor` object, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

`DataFlavor` objects support polymorphism so that if the drop target accepts `java.util.List`, and the `Transferable` object contains a `java.util.ArrayList`, the drop will succeed. Likewise, this functionality supports automatic conversion between `Arrays` and `Lists`.

If the drag and drop framework doesn't know how to represent a server `DataFlavor` object on the client component, the drop target will be configured to allow all drops to succeed on the client.

The following example shows a method that the event handler method calls (the event handler itself does nothing but call this method; it is needed because this method also needs a `String` parameter that will become the value of the `outputText` component in the `panelBox` component). This method copies an array object from the event payload and assigns it to the component that initiated the event.

```
public DnDAction handleDrop(DropEvent dropEvent)
{
    Transferable dropTransferable = dropEvent.getTransferable();
    Object[] drinks =
dropTransferable.getData(DataFlavor.OBJECT_ARRAY_FLAVOR);

    if (drinks != null)
    {
        UIComponent dropComponent = dropEvent.getDropComponent();

        // Update the specified property of the drop component with the Object[] dropped
        dropComponent.getAttributes().put("value", Arrays.toString(drinks));

        return DnDAction.COPY;
    }
    else
    {
        return DnDAction.NONE;
    }
}
```

In this example, the drop component is the `panelBox` component. Because the `panelBox` does not have a `value` attribute, you would need to call the following method to set the `text` attribute of the `panelBox`.

```
dropComponent.getAttributes().put("text", Arrays.toString(drinks));
```

6. In the Components window, from the Operations panel, drag a **Client Attribute** and drop it as a child to the source component on the page.

This tag is used to define the payload of the source for the event. Define the following for the `clientAttribute` tag in the Properties window:

- **Name:** Enter any name for the payload.

- **Value:** Enter an EL expression that evaluates to the value of the payload. In the drinks example, this would resolve to the `Array` that holds the different drink values.
7. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Attribute Drag Source** and drop it as a child to the source component on the page.

In the Insert Attribute Drag Source dialog, use the dropdown list to select the `name` defined for the `clientAttribute` tag created in the previous step. Doing so makes the value of the `clientAttribute` tag the source's payload. The following example shows the code for an `outputText` component that is the source of the drag and drop operation.

```
<af:outputText value="Drag to see drinks">
  <af:clientAttribute name="drinks" value="#{myBean.drinks}"/>
  <af:attributeDragSource attribute="drinks"/>
</af:outputText>
```

What Happens at Runtime: How to Use Keyboard Modifiers

When performing a drag and drop operation, users can press keys on the keyboard (called keyboard modifiers) to select the action they wish to take on a drag and drop. The drag and drop framework supports the following keyboard modifiers:

- SHIFT: MOVE
- CTRL: COPY
- CTRL+SHIFT: LINK

When a user executes the drag and drop operation, the drop target first determines that it can accept the drag source's data flavor value. Next, if the source and target are collections, the framework intersects the actions allowed between the drag source and drop target and executes the action (one of COPY, MOVE, or LINK) in that order from the intersection. When there is only one valid action, that action is executed. When there is more than one possible action and the user's keyboard modifier matches that choice, then that is the one that is executed. If either no keyboard modifier is used, or the keyboard modifier used does not match an allowed action, then the framework chooses COPY, MOVE, LINK in that order, from the set of allowed actions.

For example, suppose you have a drop target that supports COPY and MOVE. First the drop target determines that drag source is a valid data flavor. Next, it determines which action to perform when the user performs the drop. In this example, the set is COPY and MOVE. If the user holds down the `SHIFT` key while dragging (the keyboard modifier for MOVE), the framework would choose the MOVE action. If the user is doing anything other than holding down the `SHIFT` key when dragging, the action will be COPY because COPY is the default when no modifier key is chosen (it is first in the order). If the user is pressing the `CTRL` key, that modifier matches COPY, so COPY would be performed. If the user was pressing the `CTRL+SHIFT` keys, the action would still be COPY because that modifier matches the LINK action which is not in the intersected set of allowed actions.

 **Note:**

Because information is lost during the roundtrip between Java and JavaScript, the data in the drop may not be the type that you expect. For example, all numeric types appear as `double` objects, `char` objects appear as `String` objects, `List` and `Array` objects appear as `List` objects, and most other objects appear as `Map` objects. For more information, see [What You May Need to Know About Marshalling and Unmarshalling Data..](#)

What You May Need to Know About Using the ClientDropListener

The `dropTarget` tag contains the `clientDropListener` attribute where you can reference JavaScript that will handle the drop event on the client. The client handler should not take any parameters and return an `AdfDnDContext` action. For example, if the method returns `AdfDnDContext.ACTION_NONE` the drop operation will be canceled and no server call will be made; if the method returns `AdfDnDContext.ACTION_COPY`, a copy operation will be allowed and a server call will be made which will execute the `dropListener` method if it exists.

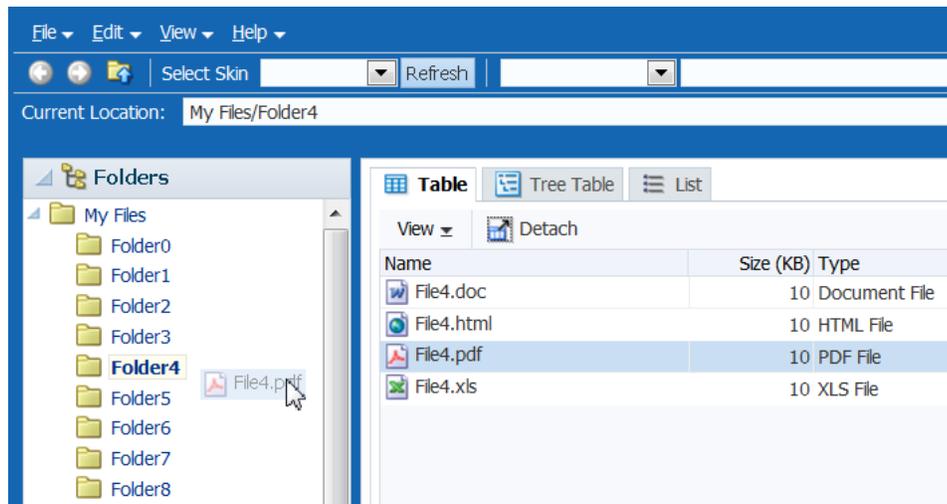
For example, suppose you want to log a message when the drop event is invoked. You might create a client handler to handle logging that message and then returning the correct action so that the server listener is invoked. The following example shows a client handler that uses the logger to print a message.

```
<script>
/**
 * Shows a message.
 */
function showMessage()
{
    AdfLogger.LOGGER.logMessage(AdfLogger.ALL, "clientDropListener handler,
        copying...");
    return AdfDnDContext.ACTION_COPY;
}
</script>
```

Adding Drag and Drop Functionality for Collections

You can enable ADF Faces drag and drop functionality for collections displayed by ADF Faces components. Drag and drop functionality that you implement for ADF Faces components supports the end-user experience of interactively dragging an object from one collection and dropping it onto a supported target collection in the user interface of the displayed web page.

You use the `collectionDropTarget` and `dragSource` tags to add drag and drop functionality that allows users to drag an item from one collection (for example, a row from a table), and drop it into another collection component such, as a tree. For example, in the File Explorer application, users can drag a file from the table that displays directory contents to any folder in the directory tree. [Figure 39-3](#) shows the `File0.doc` object being dragged from the table displaying the contents of the `Folder0` directory to the `Folder3` directory. Once the drop is complete, the object will become part of the collection that makes up `Folder3`.

Figure 39-3 Drag and Drop Functionality in the File Explorer Application

As with dragging and dropping single objects, you can have a drop on a collection with a copy, move, or copy and paste as a link (or a combination of the three), and use `dataFlavor` tags to limit what a target will accept.

When the target source is a collection and it supports the move operation, you may also want to also implement a method for the `dragDropEndListener` attribute, which is referenced from the source component and is used to clean up the collection after the drag and drop operation. See [What You May Need to Know About the dragDropEndListener](#).

How to Add Drag and Drop Functionality for Collections

To add drag and drop functionality for collections, instead of using the `dropTarget` tag, you use the `collectionDropTarget` tag. You then must implement the event handler method that will handle the logic for the drag and drop action. Next, you define the source for the drag and drop operation using the `dragSource` tag.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality for Collections](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

You will also need to create the source and target components on the page.

To add drag and drop functionality:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Collection Drop Target** and drop it as a child to the target collection component on the page.

2. In the Insert Collection Drop Target dialog, enter an expression for the `dropListener` attribute that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 4).
3. In the Properties window, set the following:
 - `actions`: Select the actions that can be performed on the source during the drag and drop operation.
If no actions are specified, the default is `COPY`.
 - `modelName`: Define the model for the collection.
The value of the `modelName` attribute is a `String` object used to identify the drag source for compatibility purposes. The value of this attribute must match the value of the `discriminant` attribute of the `dragSource` tag you will use in a Step 6. In other words, this is an arbitrary name and works when the target and the source share the same `modelName` value or `discriminant` value.
4. In the managed bean inserted into the EL expression in Step 2, implement the handler for the drop event.

This method must take a `DropEvent` event as a parameter and return a `DnDAction`. This method should use the `DropEvent` to get the `Transferable` object and from there get the `RowKeySet` (the rows that were selected for the drag). Using the `CollectionModel` obtained through the `Transferable` object, the actual `rowData` can be obtained to complete the drop. The method should then check the `DropEvent` to determine whether it will accept the drop or not. If the method accepts the drop, it should perform the drop and return the `DnDAction` it performed -- `DnDAction.COPY`, `DnDAction.MOVE` or `DnDAction.LINK`, otherwise it should return `DnDAction.NONE` to indicate that the drop was rejected.

The following example shows the event handler method on the `CollectionDnd.java` managed bean used in the `collectionDropTarget` demo that handles the copy of the row between two tables.

```
public DnDAction handleDrop(DropEvent dropEvent)
{
    return _handleDrop(dropEvent, getTargetValues(), "DnDDemoModel");
}
private DnDAction _handleDrop(DropEvent dropEvent,
                               ArrayList<DnDDemoData> targetValues,
                               String discriminator)
{
    Transferable transferable = dropEvent.getTransferable();
    // The data in the transferable is the row key for the dragged component.
    DataFlavor<RowKeySet> rowKeySetFlavor =
        DataFlavor.getDataFlavor(RowKeySet.class, discriminator);
    RowKeySet rowKeySet = transferable.getData(rowKeySetFlavor);
    if (rowKeySet != null)
    {
        // Get the model for the dragged component.
        CollectionModel dragModel = transferable.getData(CollectionModel.class);
        if (dragModel != null)
        {
            // Set the row key for this model using the row key from the transferable.
            Object currKey = rowKeySet.iterator().next();
            dragModel.setRowKey(currKey);

            // And now get the actual data from the dragged model.
```

```

// Note this won't work in a region.
//Need to change this to use collectionModel data flavor.
DnDDemoData dnDDemoData = (DnDDemoData)dragModel.getRowData();

// Put the dragged data into the target model directly.
// Note that if you wanted validation/business rules on the drop,
// this would be different.
// getTargetValues() is the target collection used by the target component
if (dropEvent.getProposedAction() == DnDAction.LINK)
    dnDDemoData = DnDDemoData.addALink(dnDDemoData);
    targetValues.add(dnDDemoData);
}
return dropEvent.getProposedAction();
}
else
{
    return DnDAction.NONE;
}
}

```

5. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop a **Drag Source** as a child to the source component.
6. With the `dragSource` tag selected, in the Properties window set the actions, discriminant, and any `dragDropEndListener` as configured for the target. For instance, the `dragSource` tag may appear similar to the following:

```

<af:dragSource actions="MOVE" discriminant="DnDDemoModel
dragDropEndListener="#{collectionDnD.endListener}"/>

```

What You May Need to Know About the `dragDropEndListener`

There may be cases when after a drop event, you have to clean up the source collection. For example, if the drag caused a move, you may have to clean up the source component so that the moved item is no longer part of the collection.

The `dragSource` tag contains the `dragDropEndListener` attribute that allows you to register a handler that contains logic for after the drag drop operation ends.

For example, if you allow a drag and drop to move an object, you may have to physically remove the object from the source component once you know the drop succeeded. The following example shows a handler for a `dragDropEndListener` attribute

```

public void endListener(DropEvent dropEvent)
{
    Transferable transferable = dropEvent.getTransferable();

    // The data in the transferrable is the row key for the dragged component.
    DataFlavor<RowKeySet> rowKeySetFlavor =
        DataFlavor.getDataFlavor(RowKeySet.class, "DnDDemoModel");
    RowKeySet rowKeySet = transferable.getData(rowKeySetFlavor);
    if (rowKeySet != null)
    {
        Integer currKey = (Integer)rowKeySet.iterator().next();

        // Remove the dragged data from the source model directly.
        // getSourceValues() represents a collection object used by the source
        // component
        Object removed = getSourceValues().remove(currKey.intValue());
    }
}

```

```

}
// Need to add the drag source table so it gets redrawn.
// The drag source component needs to be partially refreshed explicitly, while
// drop target component automatically refreshed and displayed.

AdfFacesContext.getCurrentInstance().addPartialTarget(dropEvent.getDragComponent());

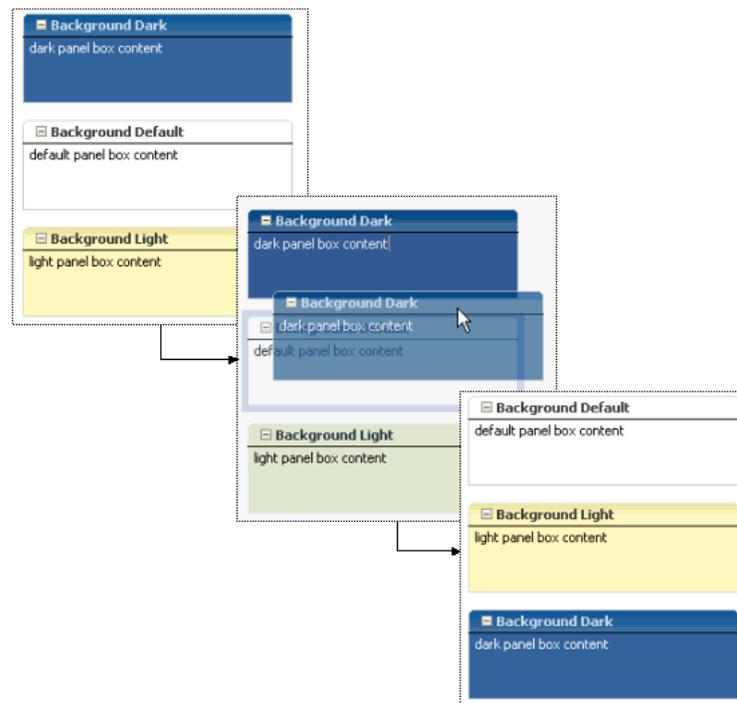
```

Adding Drag and Drop Functionality for Components

You can enable ADF Faces drag and drop functionality for entire ADF Faces components. Drag and drop functionality that you implement for ADF Faces components supports the end-user experience of interactively dragging a component and dropping it somewhere else in the user interface of the displayed web page.

You can allow components to be moved from one parent to another, or you can allow child components of a parent component to be reordered. For example, [Figure 39-4](#) shows the darker `panelBox` component being moved from being the first child component of the `panelGrid` component to the last.

Figure 39-4 Drag and Drop Functionality Between Components



Note:

If you want to move components into or out of a `panelDashboard` component, then you need to use procedures specific to that component. See [Adding Drag and Drop Functionality Into and Out of a `panelDashboard` Component](#).

How to Add Drag and Drop Functionality for Components

Adding drag and drop functionality for components is similar for objects. However, instead of using the `attributeDragSource` tag, use the `componentDragSource` tag. As with dragging and dropping objects or collections, you also must implement a `dropListener` handler.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality for Components](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

To add drag and drop functionality:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Drop Target** and drop it as a child to the target component on the page.
2. In the Insert Drop Target dialog, enter an expression that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 4).
3. With the `dropTarget` tag still selected, in the Properties window, select a valid action set for the `actions` attribute.
4. In the managed bean referenced in the EL expression created in Step 2 for the `dropListener` attribute, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

If the method accepts the drop, it should perform the drop and return the `DnDAction` it performed -- `DnDAction.COPY`, `DnDAction.MOVE` or `DnDAction.LINK`, otherwise it should return `DnDAction.NONE` to indicate that the drop was rejected

This handler method should use the `DropEvent` event to get the `Transferable` object and its data and then complete the move or copy, and reorder the components as needed. Once the method completes the drop, it should return the `DnDAction` it performed. Otherwise, it should return `DnDAction.NONE` to indicate that the drop was rejected.

The following example shows the `handleComponentMove` event handler on the `DemoDropHandler.java` managed bean used by the `componentDragSource` JSF page in the demo application.

```
public DnDAction handleComponentMove(DropEvent dropEvent)
{
    Transferable dropTransferable = dropEvent.getTransferable();
    UIComponent movedComponent = dropTransferable.getData
        (DataFlavor.UICOMPONENT_FLAVOR);
    if ((movedComponent != null) &&
        DnDAction.MOVE.equals(dropEvent.getProposedAction()))
    {
        UIComponent dropComponent = dropEvent.getDropComponent();
        UIComponent dropParent = dropComponent.getParent();
        UIComponent movedParent = movedComponent.getParent();
        UIComponent rootParent = null;
        ComponentChange change = null;
    }
}
```

```

// Build the new list of IDs, placing the moved component after the dropped
//component.
String movedLayoutId = movedParent.getId();
String dropLayoutId = dropComponent.getId();

List<String> reorderedIdList = new
    ArrayList<String>(dropParent.getChildCount());

for (UIComponent currChild : dropParent.getChildren())
{
    String currId = currChild.getId();

    if (!currId.equals(movedLayoutId))
    {
        if(!movedLayoutIdFound && currId.equals(dropLayoutId))
            reorderedIdList.add(movedLayoutId);
        reorderedIdList.add(currId);
        if(movedLayoutIdFound && currId.equals(dropLayoutId))
            reorderedIdList.add(movedLayoutId);
    }
    else
        movedLayoutIdFound = true;
}

change = new ReorderChildrenComponentChange(reorderedIdList);
rootParent = dropParent;
ChangeManager cm = RequestContext.getCurrentInstance().getChangeManager();

// add the change
cm.addComponentChange(FacesContext.getCurrentInstance(),rootParent, change);

// apply the change to the component tree immediately
change.changeComponent(rootParent);

// redraw the shared parent
AdfFacesContext.getCurrentInstance().addPartialTarget(rootParent);

return DnDAction.MOVE;
}
else
{
    return DnDAction.NONE;
}
}

```

5. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Component Drag Source** and drop it as a child of the source component on the page.

For instance, the `componentDragSource` tag may appear similar to the following:

```
<af:componentDragSource discriminant="col2"/>
```

Adding Drag and Drop Functionality Into and Out of a panelDashboard Component

You can enable ADF Faces drag and drop functionality within `panelDashboard` ADF Faces components. Drag and drop functionality that you implement for ADF Faces `panelDashboard` components supports the end-user experience of interactively

dragging a component into or out of the `panelDashboard` in the user interface of the displayed web page.

By default the `panelDashboard` component supports dragging and dropping components within itself. That is, you can reorder components in a `panelDashboard` component without needing to implement a listener or use additional tags. However, if you want to be able to drag a component into a `panelDashboard` component, or to drag a component out of a `panelDashboard` component, you do need to use tags and implement a listener. Because you would be dragging and dropping a component, you use the `componentDragSource` tag when dragging into the `panelDashboard`. However, because the `panelDashboard` already supports being a drop target, you do not need to use the `dropTarget` tag. Instead, you need to use a `dataFlavor` tag with a discriminant. The tag and discriminant notify the framework that the drop is from an external component.

Dragging a component out of a `panelDashboard` is mostly the same as dragging and dropping any other component. You use a `dropTarget` tag for the target and the `componentDragSource` tag for the source. However, you must also use the `dataFlavor` tag and a discriminant.

How to Add Drag and Drop Functionality Into a `panelDashboard` Component

Because the `panelDashboard` component has built-in drag and drop functionality used to reorder `panelBox` components within the dashboard, you need not use a `dropTarget` tag, but you do need to use a `dataFlavor` tag with a discriminant and implement the `dropListener`. In that implementation, you need to handle the reorder of the components.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality Into and Out of a `panelDashboard` Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

Before you begin:

1. Create a `panelDashboard` component. For more information, see [Arranging Contents in a Dashboard](#).
2. Create another component outside of the `panelDashboard` that contains `panelBox` components. For more information about `panelBox` components, see [How to Use the `panelBox` Component](#).

To add drag and drop functionality into a `panelDashboard` component:

1. In the Structure window, select the `panelDashboard` component that is to be the target component.
2. In the Properties window, for **DropListener**, enter an expression that evaluates to a method on a managed bean that will handle the drop event (you will create this code in Step 6).

3. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Data Flavor** and drop it as a child to the `panelDashboard` component.
4. In the Insert Data Flavor dialog, enter `javax.faces.component.UIComponent`.
5. In the Properties window, set **Discriminant** to a unique name that will identify the components allowed to be dragged into the `panelDashboard` component, for example, `dragIntoDashboard`.
6. In the Components window, from the Operations panel, in the Drag and Drop group, drag a **Component Drag Source** and drop it as a child to the `panelBox` component that will be the source component.
7. In the Properties window, set **Discriminant** to be the same value as entered for the **Discriminant** on the `panelDashboard` in Step 5.

How to Add Drag and Drop Functionality Out of a `panelDashboard` Component

Implementing drag and drop functionality out of a `panelDashboard` component is similar to standard drag and drop functionality for other components, except that you must use a `dataFlavor` tag with a discriminant.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality Into and Out of a `panelDashboard` Component](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

How to add drag and drop functionality out of a `panelDashboard` component:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop a **Drop Target** as a child to the target component.
2. In the Insert Drop Target dialog, enter an expression that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 5) and enter `javax.faces.component.UIComponent` as the **FlavorClass**.
3. With the `dropTarget` tag still selected, in the Properties window, select **MOVE** as the value `action` attribute.
4. In the Structure window, select the `dataFlavor` tag and in the Properties window, set **Discriminant** to a unique name that will identify the `panelBox` components allowed to be dragged into this component, for example, `dragOutOfDashboard`.
5. In the managed bean referenced in the EL expression created in Step 2 for the `dropListener` attribute, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

This handler method should use the `DropEvent` event to get the `Transferable` object and its data and then complete the move and reorder the components as needed. Once the method completes the drop, it should return a `DnDAction` of `NONE`.

You can use the

`dashboardComponent.prepareOptimizedEncodingOfDeletedChild()` method to animate the removal of the `panelBox` component.

The following example shows the `handleSideBarDrop` event handler and helper methods on the `oracle.adfdemo.view.layout.DemoDashboardBean.java` managed bean used by the dashboard JSF page in the demo application.

```
public DnDAction handleSideBarDrop(DropEvent e)
{
    UIComponent dragComponent = e.getDragComponent();
    UIComponent dragParent = dragComponent.getParent();
    // Ensure that the drag source is one of the items from the dashboard:
    if (dragParent.equals(_getDashboard()))
    {
        _minimize(dragComponent);
    }
    return DnDAction.NONE; // the client is already updated, so no need to
    redraw it again
}

private void _minimize(UIComponent panelBoxToMinimize)
{
    // Make this panelBox non-rendered:
    panelBoxToMinimize.setRendered(false);

    // If the dashboard is showing, let's perform an optimized render so the
    whole dashboard doesn't
    // have to be re-encoded.
    // If the dashboard is hidden (because the panelBox is maximized), we will
    not do an optimized
    // encode since we need to draw the whole thing.
    if (_maximizedPanelKey == null)
    {
        int deleteIndex = 0;
        RichPanelDashboard dashboard = _getDashboard();
        List<UIComponent> children = dashboard.getChildren();
        for (UIComponent child : children)
        {
            if (child.equals(panelBoxToMinimize))
            {
                dashboard.prepareOptimizedEncodingOfDeletedChild(
                    FacesContext.getCurrentInstance(),
                    deleteIndex);
                break;
            }
            if (child.isRendered())
            {
                // Only count rendered children since that's all that the
                panelDashboard can see:
                deleteIndex++;
            }
        }
    }
    RequestContext rc = RequestContext.getCurrentInstance();
    if (_maximizedPanelKey != null)
    {
        // Exit maximized mode:
        _maximizedPanelKey = null;
        UIXSwitcher switcher = _getSwitcher();
        switcher.setFacetName("restored");
    }
}
```

```
rc.addPartialTarget(switcher);  
}  
// Redraw the side bar so that we can update the colors of the opened items:  
rc.addPartialTarget(_getSideBarContainer());  
}
```

6. In the Components window, from the Operations panel, drag and drop a **Component Drag Source** as a child of the source `panelBox` component within the `panelDashboard` component.
7. In the Properties window, set **Discriminant** to be the same value as entered for the **Discriminant** on the `dataFlavor` tag for the target component in Step 4.

Adding Drag and Drop Functionality to a Calendar

You can enable ADF Faces drag and drop functionality for ADF Faces calendar components. Drag and drop functionality that you implement for ADF Faces calendar components supports the end-user experience of interactively dragging an activity in a calendar from one start time or date to another in the user interface of the displayed web page.

The calendar includes functionality that allows users to drag the handle of an activity to change the end time. However, if you want users to be able to drag and drop an activity to a different start time, or even a different day, then you implement drag and drop functionality. Drag and drop allows you to not only move an activity, but also to copy one.

How to Add Drag and Drop Functionality to a Calendar

You add drag and drop functionality by using the `calendarDropTarget` tag. Unlike dragging and dropping a collection, there is no need for a source tag; the target (that is the object to which the activity is being moved, in this case, the calendar) is responsible for moving the activities. If the source (that is, the item to be moved or copied), is an activity within the calendar, then you use only the `calendarDropTarget` tag. The tag expects the `Transferable` to be a `CalendarActivity` object.

However, you can also drag and drop objects from outside the calendar. When you want to enable this, use `dataFlavor` tags configured to allow the source object (which will be something other than a `calendarActivity` object) to be dropped.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [Adding Drag and Drop Functionality to a Calendar](#).

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see [Additional Functionality for Drag and Drop](#).

To add drag and drop functionality to a calendar:

1. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop a **Calendar Drop Target** as a child to the `calendar` component.
2. In the Insert Calendar Drop Target dialog, enter an expression for the `dropListener` attribute that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 4).

3. In the Properties window, set **Actions**. This value determines whether the activity (or other source) can be moved, copied, or copied as a link, or any combination of the three. If no action is specified, the default is `COPY`.
4. In the managed bean inserted into the EL expression in Step 2, implement the handler for the drop event.

This method must take a `DropEvent` event as a parameter and return a `DnDAction`. The `DnDAction` is the action that will be performed when the source is dropped. Valid return values are `COPY`, `MOVE`, and `LINK`, and are set when you define the `actions` attribute in Step 3. This method should use the `DropEvent` to get the `Transferable` object, and from there, access the `CalendarModel` object in the dragged data and from there, access the actual data. The listener can then add that data to the model for the source and then return the `DnDAction` it performed: `DnDAction.COPY`, `DnDAction.MOVE` or `DnDAction.LINK`; otherwise, the listener should return `DnDAction.NONE` to indicate that the drop was rejected.

The drop site for the drop event is an instance of the `oracle.adf.view.rich.dnd.CalendarDropSite` class. For an example of a drag and drop handler for a calendar, see the `handleDrop` method on the `oracle.adfdemo.view.calendar.rich.DemoCalendarBean` managed bean in the ADF Faces Components Demo application.

5. If the source for the activity is external to the calendar, drag a **Data Flavor** and drop it as a child to the `calendarDropTarget` tag. This tag determines the type of object that can be dropped onto the target, for example a `String` or a `Date` object. Multiple `dataFlavor` tags are allowed under a single drop target to allow the drop target to accept any of those types.
6. In the Insert Data Flavor dialog, enter the class for the object that can be dropped onto the target, for example `java.lang.Object`.

 **Tip:**

To specify a typed array in a `dataFlavor` tag, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

What You May Need to Know About Dragging and Dropping in a Calendar

For dragging and dropping activities within a calendar, users can drag and drop only within a view. That is, users can drag an activity from one time slot to another in the day view, but cannot cut an activity from a day view and paste it into a month view.

When the user is dragging and dropping activities in the day or week view, the calendar marks the drop site by half-hour increments. The user cannot move any all-day or multi-day activities in the day view.

In the week view, users can move all-day and multi-day activities, however, they can be dropped only within other all-day slots. That is, the user cannot change an all-day activity to an activity with start and end times. In the month view, users can move all-day and multi-day activities to any other day.

Adding Drag and Drop Functionality for DVT Components

You can enable ADF Faces drag and drop functionality for a variety of Data Visualization Technology (DVT) components. Drag and drop functionality that you implement for ADF Faces DVT components supports the end-user experience of interactively dragging elements of DVT components or the components themselves and dropping them within the user interface of the displayed web page.

You can configure drag and drop functionality for the following DVT components:

- Pareto and stock graphs (drop target only)
- Gantt charts
- Hierarchy viewers
- Sunbursts
- Thematic Maps
- Timelines
- Treemaps

DVT components use essentially the same process as dragging and dropping other ADF Faces components. However, DVT components may impose limitations on the items that you can drag to or drop from the component.

As with dragging and dropping objects or collections, you must also implement a `dropListener` handler to respond to the drop requests. The object of the drop event is called the `Transferable`, which contains the payload of the drop. Your listener must access the `Transferable` object, and from there, use the `DataFlavor` object to verify that the object can be dropped. You then use the drop event to get the target component and update the property with the dropped object.

Adding Drop Functionality for DVT Pareto and Stock Graphs

Pareto and stock graphs can be configured as a drop target to allow drops from other ADF components. For example, you can configure a stock graph to allow drops from an ADF table cell.

How to Add Drop Functionality to Pareto and Stock Graphs

To configure a Pareto or stock graph as a drop target, add the `af:dropTarget` tag as a child of the Pareto or stock Graph, and add a method in a managed bean to respond to the drop event. The following example shows a sample drop listener for a graph configured to accept drops from an ADF table.

You must also configure the ADF Faces component, object, or collection as a drag source and define the method that will respond to the drag.

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [About Drag and Drop Functionality](#).

You must complete the following tasks:

- Add a Pareto or stock graph to your page.

For help with creating the DVT components, see [Introduction to ADF Data Visualization Components](#).

- If you plan to allow drops to the DVT component, add the component that will serve as the drag source to the page.

For help with adding other ADF Faces components, see [ADF Faces Components](#).

- Create the method that will listen for drops on the graph. For information about using managed beans, see [Creating and Using Managed Beans](#).

To configure a Pareto or stock graph as a drop target:

1. In the Structure window, right-click `dvt:paretoGraph` or `dvt:stockGraph` and choose **Insert Inside (Pareto or Stock) > Drop Target**.
2. In the Insert Drop Target dialog, specify the **DropListener** as an EL Expression that evaluates the reference to the `oracle.adf.view.rich.event.DropEvent` method called when a drop occurs on the component.

To specify the **DropListener** used in [How to Add Drop Functionality to Pareto and Stock Graphs](#), enter: `{dragAndDrop.fromTableDropListener}`.

3. In the Insert Data Flavors dialog, specify the **flavorClass**, the fully qualified Java class name for this `dataFlavor`. If the drop contains this `dataFlavor`, the drop target is guaranteed to be able to retrieve an Object from the drop with this Java type using this `dataFlavor`.

For example, to specify the **flavorClass** for a drop target configured to allow drops from an ADF table, enter: `org.apache.myfaces.trinidad.model.RowKeySet`.

Example 39-1 Managed Bean Sample for Handling Drag and Drop Target

```
public class dragAndDrop {
    public DnDAction fromTableDropListener(DropEvent event) {
        Transferable transferable = event.getTransferable();
        DataFlavor<RowKeySet> dataFlavor = DataFlavor.getDataFlavor(RowKeySet.class,
"fromTable");
        RowKeySet set = transferable.getData(dataFlavor);
        Employee emp = null;
        if(set != null && !set.isEmpty()) {
            int index = (Integer) set.iterator().next();
            emp = m_tableModel.get(index);
        }
        if(emp == null)
            return DnDAction.NONE;
        DnDAction proposedAction = event.getProposedAction();
        if(proposedAction == DnDAction.COPY) {
            m_graphList.add(emp);
        }
        else if(proposedAction == DnDAction.LINK) {
            m_graphList.add(emp);
        }
        else if(proposedAction == DnDAction.MOVE) {
            m_graphList.add(emp);
            m_tableModel.remove(emp);
        }
        else
            return DnDAction.NONE;
        RequestContext.getCurrentInstance().addPartialTarget(event.getDragComponent());
        return event.getProposedAction();
    }
}
```

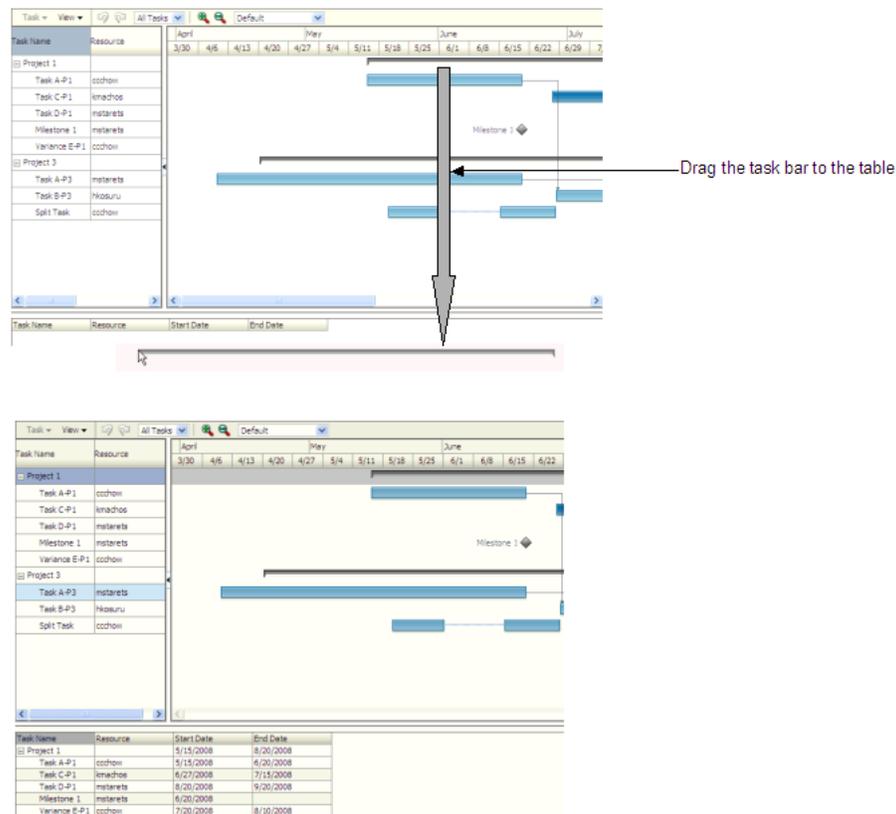
You also must implement a listener for the drop event. The object of the drop event is called the `Transferable`, which contains the payload of the drop. Your listener must access the `Transferable` object, and from there, use the `DataFlavor` object to verify that the object can be dropped. You then use the drop event to get the target component and update the property with the dropped object.

Adding Drag and Drop Functionality for DVT Gantt Charts

When you want users to be able to drag and drop between Gantt charts and other components, you use the `dragSource` and `dropTarget` tags. Additionally, you use the `DataFlavor` object to determine the valid Java types of sources for the drop target. You also must implement any required functionality in response to the drag and drop action. Both the `projectGantt` and `schedulingGantt` components support drag and drop functionality.

For example, suppose you have an `projectGantt` component and you want the user to be able to drag one time bucket to a `treeTable` component and have that component display information about the time bucket, as shown in [Figure 39-5](#).

Figure 39-5 Dragging and Dropping an Object



The `projectGantt` component contains a `dragSource` tag. And because the user will drag the whole object and not just the `String` value of the output text that is displayed, you use the `dropTarget` tag instead of the `attributeDropTarget` tag.

You also use a `dataFlavor` tag to determine the type of object being dropped. On this tag, you can define a discriminant value. This is helpful if you have two targets and two sources, all with the same object type. By creating a discriminant value, you can be sure that each target will accept only valid sources. For example, suppose you have two targets that both accept an `TaskDragInfo` object, `TargetA` and `TargetB`. Suppose you also have two sources, both of which are `TaskDragInfo` objects. By setting a discriminant value on `TargetA` with a value of `alpha`, only the `TaskDragInfo` source that provides the discriminant value of `alpha` will be accepted.

You also must implement a listener for the drop event. The object of the drop event is called the `Transferable`, which contains the payload of the drop. Your listener must access the `Transferable` object, and from there, use the `DataFlavor` object to verify that the object can be dropped. You then use the drop event to get the target component and update the property with the dropped object.

How to Add Drag and Drop Functionality for a DVT Gantt Component

To add drag and drop functionality, first add tags to a component that define it as a target for a drag and drop action. Then implement the event handler method that will handle the logic for the drag and drop action. Last, you define the sources for the drag and drop. For information about what happens at runtime, see [What Happens at Runtime: How to Use Keyboard Modifiers](#). For information about using the `clientDropListener` attribute, see [What You May Need to Know About Using the ClientDropListener](#).

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [About Drag and Drop Functionality](#).

You must complete the following tasks:

- Add the DVT component to your page.
For help with creating the DVT components, see [Introduction to ADF Data Visualization Components](#).
- If you plan to allow drops to the DVT component, add the component that will serve as the drag source to the page.
For help with adding other ADF Faces components, see [ADF Faces Components](#).
- If you plan on allowing drags from the DVT component to another component, add the component that will serve as the drop target to the page.

To add drag and drop functionality:

1. In the Components window, from the Operations panel, drag a **Drop Target** tag and drop it as a child to the target component.
2. In the Insert Drop Target dialog, enter an expression that evaluates to a method on a managed bean that will handle the event (you will create this code in Step 6).

Tip:

You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see [What You May Need to Know About Using the ClientDropListener](#).

3. In the Insert Data Flavor dialog, enter the class for the object that can be dropped onto the target, for example `java.lang.Object`. This selection will be used to create a `dataFlavor` tag, which determines the type of object that can be dropped onto the target. Multiple `dataFlavor` tags are allowed under a single drop target to allow the drop target to accept any of those types.

 **Tip:**

To specify a typed array in a `DataFlavor` tag, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

4. In the Properties window, set a value for **Discriminant**, if needed. A discriminant is an arbitrary string used to determine what sources of the type specified by the `dataFlavor` will be allowed as a source.
5. In the Structure window, select the `dropTarget` tag. In the Properties window, select a value for **Actions**. This defines what actions are supported by the drop target. Valid values can be `COPY` (copy and paste), `MOVE` (cut and paste), and `LINK` (copy and paste as a link), for example:

```
MOVE COPY
```

If no actions are specified, the default is `COPY`.

The following example shows the code for a `dropTarget` component that takes a `TaskDragInfo` object as a drop source. Note that because `COPY` was set as the value for the `actions` attribute, that will be the only allowed action.

```
<af:treeTable id="treeTableDropTarget"
    var="task" value="#{projectGanttDragSource.treeTableModel}">
  <f:facet name="nodeStamp">
    <af:column headerText="Task Name">
      <af:outputText value="#{task.taskName}"/>
    </af:column>
  </f:facet>
  <af:column headerText="Resource">
    <af:outputText value="#{task.resourceName}"/>
  </af:column>
  <af:column headerText="Start Date">
    <af:outputText value="#{task.startTime}"/>
  </af:column>
  <af:column headerText="End Date">
    <af:outputText value="#{task.endTime}"/>
  </af:column>
  <af:dropTarget actions="COPY"
    dropListener="#{projectGanttDragSource.onTableDrop}">
    <af:dataFlavor flavorClass=
      "oracle.adf.view.faces.bi.component.gantt.TaskDragInfo"/>
  </af:dropTarget>
</af:treeTable>
```

6. In the managed bean referenced in the EL expression created in Step 2, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

This method must take a `DropEvent` event as a parameter and return a `DnDAction` object, which is the action that will be performed when the source is dropped. Valid return values are `DnDAction.COPY`, `DnDAction.MOVE`, and `DnDAction.LINK`, and

were set when you defined the target attribute in Step 5. This method should check the `DropEvent` event to determine whether or not it will accept the drop. If the method accepts the drop, it should perform the drop and return the `DnDAction` object it performed. Otherwise, it should return `DnDAction.NONE` to indicate that the drop was rejected.

The method must also check for the presence for each `dataFlavor` object in preference order.

 **Tip:**

If your target has more than one defined `dataFlavor` object, then you can use the `Transferable.getSuitableTransferData()` method, which returns a `List` of `TransferData` objects available in the `Transferable` object in order, from highest suitability to lowest.

The `DataFlavor` object defines the type of data being dropped, for example `java.lang.Object`, and must be as defined in the `DataFlavor` tag on the JSP, as created in Step 3.

 **Tip:**

To specify a typed array in a `DataFlavor` object, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

`DataFlavor` objects support polymorphism so that if the drop target accepts `java.util.List`, and the `Transferable` object contains a `java.util.ArrayList`, the drop will succeed. Likewise, this functionality supports automatic conversion between `Arrays` and `Lists`.

If the drag and drop framework doesn't know how to represent a server `DataFlavor` object on the client component, the drop target will be configured to allow all drops to succeed on the client.

The following example shows a handler method that copies a `TaskDragInfo` object from the event payload and assigns it to the component that initiated the event.

```
public DnDAction onTableDrop(DropEvent evt)
{
    // retrieve the information about the task dragged
    DataFlavor<TaskDragInfo> _flv = DataFlavor.getDataFlavor(TaskDragInfo.class,
null);
    Transferable _transferable = evt.getTransferable();

    // if there is no data in the Transferable, then the drop is unsuccessful
    TaskDragInfo _info = _transferable.getData(_flv);
    if (_info == null)
        return DnDAction.NONE;

    // find the task
    Task _draggedTask = findTask(_info.getTaskId());
    if (_draggedTask != null) {
        // process the dragged task here and indicate the drop is successful by
returning DnDAction.COPY
    }
}
```

```

        return DnDAction.COPY;
    }
    else
    return DnDAction.NONE;
}

```

7. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the source component.
8. With the `dragSource` tag selected, in the Properties window, set the allowed Actions and any needed discriminant, as configured for the target.

Adding Drag and Drop Functionality for DVT Hierarchy Viewers, Sunbursts, and Treemaps

You can configure hierarchy viewers, sunbursts, and treemaps as drag sources and drop targets for drag and drop operations between supported components on a page.

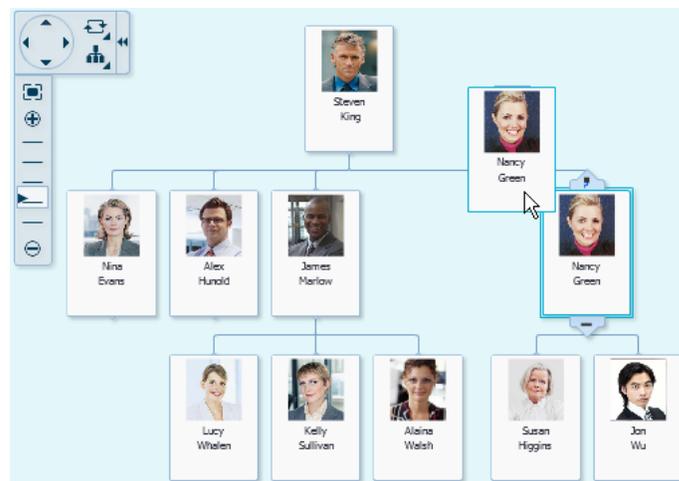
Drag and Drop Example for DVT Hierarchy Viewers

Hierarchy viewers support the following drag and drop operations:

- Drag and drop one or more nodes within a hierarchy viewer
- Drag one or more nodes from a hierarchy viewer to another component
- Drag one or more items from another component to a hierarchy viewer

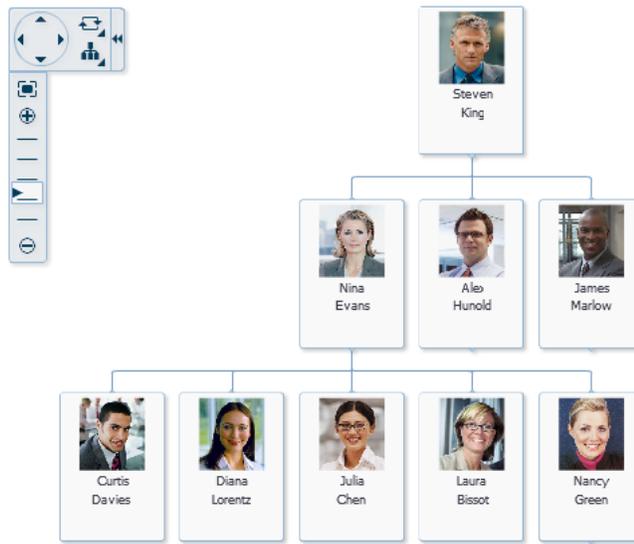
[Figure 39-6](#) shows a hierarchy viewer configured to allow drags and drops within itself. If you click and hold a node for more than one-half second, you can drag it to the background to make it another root in the hierarchy or drag it to another node to add it as a child of that node.

Figure 39-6 Hierarchy Viewer Showing a Node Drag



In this example, if you drag the node to another node, the dragged node and its children become the child of the targeted node. [Figure 39-7](#) shows the result of the drag to the node containing the data for Nina Evans. Nancy Green and her subordinates are now shown as subordinates to Nina Evans.

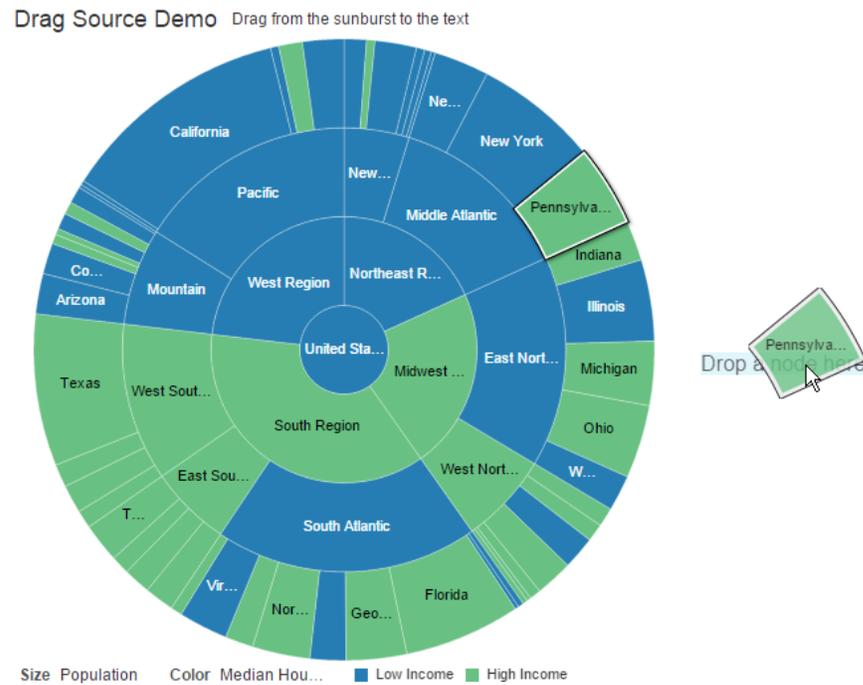
Figure 39-7 Hierarchy Viewer After Node Drag to Another Node



Drag and Drop Example for DVT Sunbursts

Sunbursts support the drag of one or more nodes to another component. The payload of the drag is a `org.apache.myfaces.trinidad.model.RowKeySet`. You can also configure sunbursts to accept drops from another object.

[Figure 39-8](#) shows a sunburst configured to allow drags from it to an `af:outputFormatted` component. If the sunburst is configured for multiple selection, the user can drag multiple nodes using the `Ctrl+click` operation.

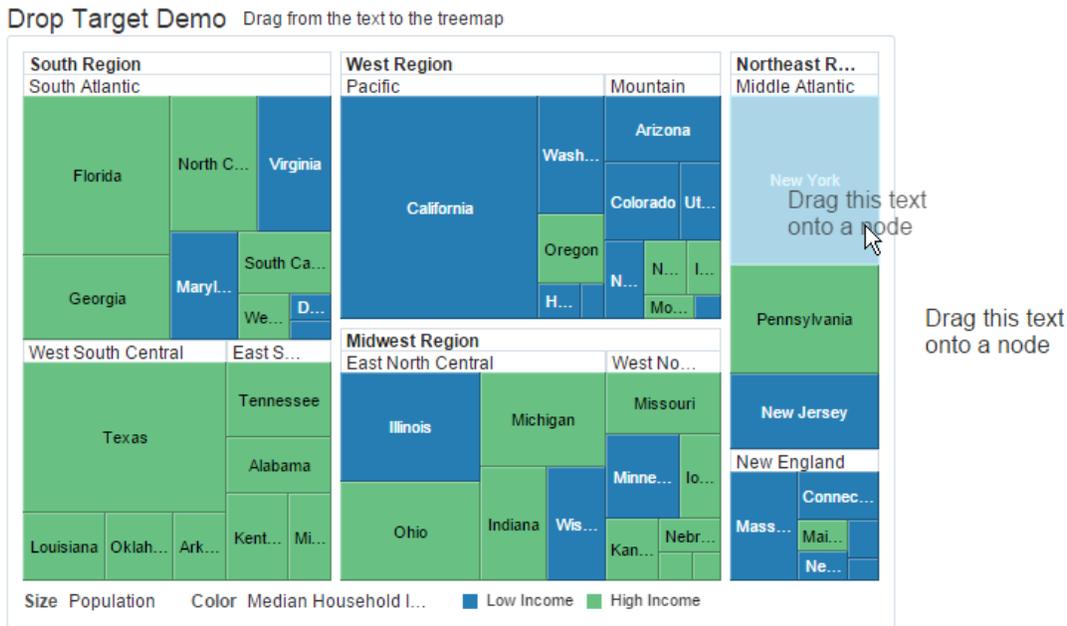
Figure 39-8 Sunburst Configured as a Drag Source

Drag and Drop Example for DVT Treemaps

Treemaps support the drag of one or more nodes to another component. The payload of the drag is a `org.apache.myfaces.trinidad.model.RowKeySet`. You can also configure treemaps to accept drops from another object.

[Figure 39-9](#) shows a treemap configured as a drop target. In this example, the drag source is an `af:outputFormatted` component.

Figure 39-9 Treemap Configured as a Drop Target



How to Add Drag and Drop Functionality for a DVT Hierarchy Viewer, Sunburst, or Treemap Component

To add drag and drop functionality, first add tags to a supported DVT component that define it as a target for a drag and drop action. Then implement the event handler method that will handle the logic for the drag and drop action. Last, you define the sources for the drag and drop. For information about what happens at runtime, see [What Happens at Runtime: How to Use Keyboard Modifiers](#). For information about using the `clientDropListener` attribute, see [What You May Need to Know About Using the ClientDropListener](#).

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [About Drag and Drop Functionality](#).

You must complete the following tasks:

- Add the DVT component to your page.
For help with creating the DVT components, see [Introduction to ADF Data Visualization Components](#).
- If you plan to allow drops to the DVT component, add the component that will serve as the drag source to the page.
For help with adding other ADF Faces components, see [ADF Faces Components](#).
- If you plan on allowing drags from the DVT component to another component, add the component that will serve as the drop target to the page.

To add drag and drop functionality to a DVT hierarchy viewer, sunburst, or treemap component:

1. To configure the DVT component as a drop target, do the following:
 - a. In the Components window, from the Operations panel, drag a **Drop Target** tag and drop it as a child to a DVT component that supports drag and drop.
 - b. In the Insert Drop Target dialog, enter an expression that evaluates to a drop listener method on a managed bean that will handle the event (you will create this code in Step 1.f).

 **Tip:**

You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see [What You May Need to Know About Using the ClientDropListener..](#)

- c. In the Insert Data Flavor dialog, enter the class for the object that can be dropped onto the target, for example `java.lang.Object`. This selection will be used to create a `dataFlavor` tag, which determines the type of object that can be dropped onto the target. Multiple `dataFlavor` tags are allowed under a single drop target to allow the drop target to accept any of those types.

 **Tip:**

To specify a typed array in a `DataFlavor` tag, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

- d. In the Properties window, set a value for **Discriminant**, if needed. A discriminant is an arbitrary string used to determine which source can drop on the target. For example, suppose you have two treemaps that both accept a `java.lang.Object`, `Treemap A` and `Treemap B`. You also have two sources, both of which are `java.lang.Object` objects. By setting a discriminant value on `GraphA` with a value of `alpha`, only the `java.lang.Object` source that provides the discriminant value of `alpha` will be accepted.
- e. In the Structure window, select the `dropTarget` tag. In the Properties window, select a value for **Actions**. This defines what actions are supported by the drop target. Valid values can be `COPY` (copy and paste), `MOVE` (cut and paste), and `LINK` (copy and paste as a link), for example:.

`MOVE COPY`

If no actions are specified, the default is `COPY`.

The following example shows the code for a treemap component that accepts a `java.lang.Object` as a drag source. Note that because `COPY` was set as the value for **Actions**, that will be the only allowed action.

```
<dvt:treemap id="t1" value="#{treemap.censusData}" var="row"
  displayLevelsChildren="3" colorLabel="Median Household Income
  sizeLabel="Population" summary="Treemap Configured as Drag
Source"
  legendSource="ag1">
  <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}">
    <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
      label="#{row.income > 50000 ? 'High Income' : 'Low
```

```
Income' }"
        type="color"/>
    </dvt:treemapNode>
    <af:dropTarget dropListener="#{treemap.toDropListener}"
        actions="COPY">
        <af:dataFlavor flavorClass="java.lang.Object"/>
    </af:dropTarget>
</dvt:treemap>
```

- f. In the managed bean referenced in the EL expression created in Step 1.b, create the event handler method (using the same name as in the EL expression) that will handle the drag and drop functionality.

This method must take a `DropEvent` event as a parameter and return a `DnDAction` object, which is the action that will be performed when the source is dropped. Valid return values are `DnDAction.COPY`, `DnDAction.MOVE`, and `DnDAction.LINK`, and were set when you defined the target attribute in Step 1.e. This method should check the `DropEvent` event to determine whether or not it will accept the drop. If the method accepts the drop, it should perform the drop and return the `DnDAction` object it performed. Otherwise, it should return `DnDAction.NONE` to indicate that the drop was rejected.

The method must also check for the presence for each `dataFlavor` object in preference order.

Tip:

If your target has more than one defined `dataFlavor` object, then you can use the `Transferable.getSuitableTransferData()` method, which returns a `List` of `TransferData` objects available in the `Transferable` object in order, from highest suitability to lowest.

The `DataFlavor` object defines the type of data being dropped, for example `java.lang.Object`, and must be as defined in the `DataFlavor` tag on the JSP, as created in Step 1.c.

Tip:

To specify a typed array in a `DataFlavor` object, add brackets (`[]`) to the class name, for example, `java.lang.Object[]`.

`DataFlavor` objects support polymorphism so that if the drop target accepts `java.util.List`, and the `Transferable` object contains a `java.util.ArrayList`, the drop will succeed. Likewise, this functionality supports automatic conversion between `Arrays` and `Lists`.

If the drag and drop framework doesn't know how to represent a server `DataFlavor` object on the client component, the drop target will be configured to allow all drops to succeed on the client.

The following example shows a handler method that copies a `java.lang.Object` from the event payload and assigns it to the component that initiated the event.

```
// imports needed by methods
import java.util.Map;
import oracle.adf.view.rich.dnd.DnDAction;
import oracle.adf.view.rich.event.DropEvent;
import oracle.adf.view.rich.datatransfer.DataFlavor;
import oracle.adf.view.rich.datatransfer.Transferable;
import org.apache.myfaces.trinidad.context.RequestContext;
import org.apache.myfaces.trinidad.render.ClientRowKeyManager;
import javax.faces.context.FacesContext;
import oracle.adf.view.faces.bi.component.treemap.UITreemap;
import javax.faces.component.UIComponent;
// variables need by methods
private String dragText = "Drag this text onto a node";
// drop listener
public DnDAction toDropListener(DropEvent event) {
    Transferable transferable = event.getTransferable();
    DataFlavor<Object> dataFlavor = DataFlavor.getDataFlavor(Object.class);
    Object transferableObj = transferable.getData(dataFlavor);
    if(transferableObj == null)
        return DnDAction.NONE;
    // Build up the string that reports the drop information
    StringBuilder sb = new StringBuilder();
    // Start with the proposed action
    sb.append("Drag Operation: ");
    DnDAction proposedAction = event.getProposedAction();
    if(proposedAction == DnDAction.COPY) {
        sb.append("Copy<br>");
    }
    else if(proposedAction == DnDAction.LINK) {
        sb.append("Link<br>");
    }
    else if(proposedAction == DnDAction.MOVE) {
        sb.append("Move<br>");
    }
    // Then add the rowKeys of the nodes that were dragged
    UIComponent dropComponent = event.getDropComponent();
    Object dropSite = event.getDropSite();
    if(dropSite instanceof Map) {
        String clientRowKey = (String) ((Map) dropSite).get("clientRowKey");
        Object rowKey = getRowKey(dropComponent, clientRowKey);
        if(rowKey != null) {
            sb.append("Drop Site: ");
            sb.append(getLabel(dropComponent, rowKey));
        }
    }
    // Update the output text
    this.dragText = sb.toString();

    RequestContext.getCurrentInstance().addPartialTarget(event.getDragComponent());
};
return event.getProposedAction();
}

public String getDragText() {
    return dragText;
}
}
```

```

private String getLabel(UIComponent component, Object rowKey) {
    if(component instanceof UITreemap) {
        UITreemap treemap = (UITreemap) component;
        TreeNode rowData = (TreeNode) treemap.getRowData(rowKey);
        return rowData.getText();
    }
    return null;
}

private Object getRowKey(UIComponent component, String clientRowKey) {
    if(component instanceof UITreemap) {
        UITreemap treemap = (UITreemap) component;
        ClientRowKeyManager crkm = treemap.getClientRowKeyManager();
        return crkm.getRowKey(FacesContext.getCurrentInstance(), component,
            clientRowKey);
    }
    return null;
}

```

2. To configure the DVT component as a drag source, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the DVT component.
 - b. With the `dragSource` tag selected, in the Properties window, set the allowed Actions and any needed discriminant, as configured for the target.

The following example shows the JSP code for a treemap configured as a drag source. Note that all actions (`COPY`, `MOVE`, and `LINK`) are permitted.

```

<dvt:treemap id="t1" value="#{treemap.censusData}" var="row"
    displayLevelsChildren="3" colorLabel="Median Household Income
    sizeLabel="Population" summary="Treemap Configured as Drag Source"
    legendSource="ag1">
    <dvt:treemapNode id="tn1" value="#{row.size}" label="#{row.text}">
        <dvt:attributeGroups id="ag1" value="#{row.income > 50000}"
            label="#{row.income > 50000 ? 'High Income' : 'Low Income'}"
            type="color"/>
    </dvt:treemapNode>
    <af:dragSource defaultAction="MOVE" actions="COPY MOVE LINK"/>
</dvt:treemap>

```

3. To use the DVT component as the drop target which will allow drags to it from another component, in the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the component that will be the source of the drag.

For example, drag and drop a **Drag Source** as a child to an `af:outputFormatted` component to display node information about a treemap. With the `dragSource` tag selected, in the Properties window, set the allowed Actions and any needed discriminant for the target.

4. To add the DVT component as a drag source for another supported DVT or ADF Faces component, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drop Target** onto the component that will receive the drop.
For example, drag and drop a **Drop Target** onto a `treeTable` component.
 - b. In the Insert Drop Target dialog, enter the name of a drop listener that the component will use to respond to the DVT component drop.
See the examples in this chapter for sample listeners.

- c. In the Insert Data Flavor dialog, enter the object that the drop target will accept. Alternatively, use the dropdown menu to navigate through the object hierarchies and choose the desired object.

For example, if you want the user to be able to drag a treemap node to a `treeTable` component and have that component display information about the treemap, enter the following for the data flavor:

```
org.apache.myfaces.trinidad.model.RowKeySet.
```

- d. In the Structure window, right-click the `af:dropTarget` component and choose **Go to Properties**.
- e. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.

The following example shows the sample code for an `af:outputFormatted` component configured to allow dragging from a treemap.

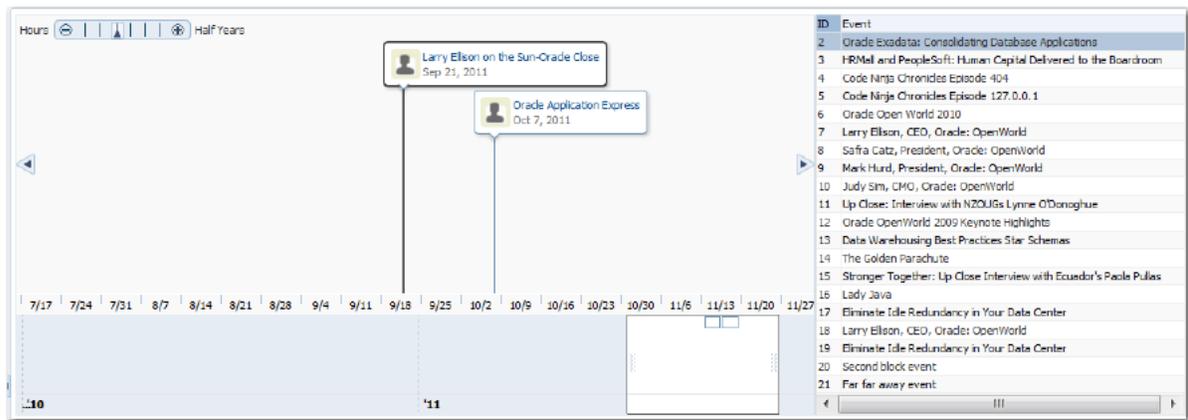
```
<af:outputFormatted value="#{treemap.dropText}" id="of1">
  <af:dropTarget dropListener="#{treemap.fromDropListener}">
    <af:dataFlavor
      flavorClass="org.apache.myfaces.trinidad.model.RowKeySet" />
    </af:dropTarget>
  </af:outputFormatted>
```

Adding Drag and Drop Functionality for Timeline Components

You can configure timelines as a drop target or drag source between collection components on a page. For example, you can drag an item from one collection such as a row from a table, and drop it into a timeline, or drag an event from a timeline and drop it into a table.

Figure 39-10 shows a timeline configured to allow drags and drops between the events in a timeline and a row in a table.

Figure 39-10 Timeline Configured for Drag and Drop Between a Table



To add drag and drop functionality, first add tags to a supported DVT component that define it as a target for a drag and drop action. Then implement the event handler method that will handle the logic for the drag and drop action. Last, you define the

sources for the drag and drop. For information about what happens at runtime, see [What Happens at Runtime: How to Use Keyboard Modifiers](#). For information about using the `clientDropListener` attribute, see [What You May Need to Know About Using the ClientDropListener](#).

Before you begin:

It may be helpful to have an understanding of drag and drop functionality. For more information, see [About Drag and Drop Functionality](#).

You must complete the following tasks:

- Add the DVT component to your page.
For help with creating the DVT components, see [Introduction to ADF Data Visualization Components](#).
- If you plan to allow drops to the DVT component, add the component that will serve as the drag source to the page.
For help with adding other ADF Faces components, see [ADF Faces Components](#).
- If you plan on allowing drags from the DVT component to another component, add the component that will serve as the drop target to the page.

To add drag and drop support to a timeline:

1. In the Structure window, right-click the `timeline` component, and select **Insert Inside Timeline > Drop Target**.
2. In the **Insert Drop Target** dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the timeline's managed bean. Alternatively, use the dropdown menu to choose **Expression Builder** and enter an EL Expression for the drop listener.

For example, to add a method named `handleDropOnTimeline()` on a managed bean named `dnd`, choose **Edit**, select **dnd** from the dropdown menu, and click **New** on the right of the **Method** field to create the `handleDropOnTimeline()` method.

The following example shows the sample drop listener and supporting methods for the timeline displayed in [Figure 39-10](#).

```
// imports needed by methods
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.bean.RequestScoped;
import oracle.adf.view.rich.datatransfer.DataFlavor;
import oracle.adf.view.rich.datatransfer.Transferable;
import oracle.adf.view.rich.dnd.DnDAction;
import oracle.adf.view.rich.event.DropEvent;
import org.apache.myfaces.trinidad.context.RequestContext;
import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.ModelUtils;
import org.apache.myfaces.trinidad.model.RowKeySet;
// drop listener
public DnDAction handleDropOnTimeline(DropEvent event)
{
```

```

Date _date = (Date)event.getDropSite();
Transferable _transferable = event.getTransferable();
RowKeySet _rowKeySet = _transferable.getData(DataFlavor.ROW_KEY_SET_FLAVOR);
Object _rowKey = _rowKeySet.iterator().next();
EmpEvent _event = (EmpEvent)m_tableModel.getRowData(_rowKey);
_event.setDate(_date);
orderInsert(_event);
RequestContext.getCurrentInstance().addPartialTarget
    (event.getDragComponent());
return DnDAction.COPY;
}
}
private void orderInsert(EmpEvent event)
{
    int _index = -1;
    ArrayList _list = (ArrayList)m_timelineModel.getWrappedData();
    for (int i=0; i<_list.size(); i++)
    {
        EmpEvent _current = (EmpEvent)_list.get(i);
        if (event.getDate().before(_current.getDate()))
        {
            _index = i;
            break;
        }
    }
    if (_index == -1)
        _list.add(event);
    else
        _list.add(_index, event);
    ArrayList _list2 = (ArrayList)m_tableModel.getWrappedData();
    _list2.remove(event);
}
}

```

3. Click **OK**, and in the **Insert Data Flavor** dialog, enter `org.apache.myfaces.trinidad.model.RowKeySet`.
4. In the Structure window, right-click the `af:dropTarget` component and choose **Go to Properties** to set the following attributes in the Properties window:
 - **Actions:** Enter a list of the operations that the drop target will accept, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drop target will use `COPY`.
 - **Discriminant:** Specify the model name shared by the drop target and drag source for compatibility purposes. The value of this attribute must match the value of the `discriminant` attribute of the `af:dragSource` component you will set for the collection component receiving the drags from the timeline.
5. To configure another collection component as the drag source for drops into the timeline, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Drag Source** tag as a child to the component that will be the source of the drag.
For example, drag and drop a **Drag Source** tag as a child to an `af:table` component.
 - b. In the Properties window, for the component's **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces.
 - c. For the component's **Discriminant** field, specify the model name shared by the drop target and drag source for compatibility purposes.

6. To configure the timeline as a drag source, in the Components window, from the Operations panel, drag and drop a **Drag Source** tag as a child to the timeline.
7. In the Structure window, right-click the `af:dragSource` component and choose **Go to Properties** to set the following attributes in the Properties window:
 - **Actions:** Enter a list of the operations that the collection drop target component will accept, separated by spaces.
 - **Discriminant:** Specify the name of the model shared by the drag source and collection drop target for compatibility purposes. The value of this attribute must match the value of the `modelName` attribute of the `af:collectionDropTarget` component you will set for the collection component receiving the drags from the timeline.
8. To make another collection component the drop target for drops from the timeline, do the following:
 - a. In the Components window, from the Operations panel, drag and drop a **Collection Drop Target** onto the component that will receive the drop.

For example, drag and drop a **Collection Drop Target** as a child to an `af:table` component that displays the results of the drop.
 - b. In the **Insert Drop Target** dialog, enter the name of the drop listener or use the dropdown menu to choose **Edit** to add a drop listener method to the appropriate managed bean.

The following example shows the sample drop listener for the timeline displayed in [Figure 39-10](#). This example uses the same imports and helper methods used in the previous example, and they are not included here.

```
//Drop Listener
public DnDAction handleDropOnTable(DropEvent event)
{
    Integer _dropSite = (Integer)event.getDropSite();
    Transferable _transferable = event.getTransferable();
    RowKeySet _rowKeySet =
_transferferable.getData(DataFlavor.ROW_KEY_SET_FLAVOR);
    Object _rowKey = _rowKeySet.iterator().next();
    EmpEvent _event = (EmpEvent)m_timelineModel.getRowData(_rowKey);
    ArrayList _list = (ArrayList)m_tableModel.getWrappedData();
    _list.add(_dropSite.intValue(), _event);
    ArrayList _list2 = (ArrayList)m_timelineModel.getWrappedData();
    _list2.remove(_event);
    RequestContext.getCurrentInstance().addPartialTarget
        (event.getDragComponent());
    return DnDAction.COPY;
}
private static Date parseDate(String date)
{
    Date ret = null;
    try
    {
        ret = s_format.parse(date);
    }
    catch (ParseException e)
    {
        e.printStackTrace();
    }
    return ret;
}
```

- c. Click OK, and in the **Insert Data Flavor** dialog, enter `org.apache.myfaces.trinidad.model.RowKeySet`.
- d. In the Structure window, right-click the `af:dropTarget` component and choose **Go to Properties**.
- e. In the Properties window, in the **Actions** field, enter a list of the operations that the drop target will accept, separated by spaces.
- f. In the **ModelName** field, define the model for the collection. The value of the `modelName` attribute is a String object used to identify the drag source for compatibility purposes. The value of this attribute must match the value of the `discriminant` attribute of the `af:dragSource` component.

The following example shows the JSF page sample code for the ADF Faces Components Demo application illustrated in [Figure 39-10](#). For additional information about the `af:table` component, see [Using Tables, Trees, and Other Collection-Based Components](#).

```
<dvt:timeline id="t11" startTime="2010-01-01" endTime="2011-12-31"
  inlineStyle="width:800px;height:400px" itemSelection="single">
  <f:attribute name="horizontalFetchSizeOverride" value="3000"/>
  <dvt:timelineSeries id="ts1" var="evt" value="{dnd.timelineModel}">
    <dvt:timelineItem id="ti1" value="{evt.date}" group="{evt.group}">
      <af:panelGroupLayout id="pg1" layout="horizontal">
        <af:image id="img1" inlineStyle="width:30px;height:30px"
          source="/resources/images/timeline/employment.png"/>
        <af:spacer width="3"/>
        <af:panelGroupLayout id="pg2" layout="vertical">
          <af:outputText id="ot1" inlineStyle="color:#084B8A"
            value="{evt.description}" noWrap="true"/>
          <af:outputText id="ot2" value="{evt.date}"
            inlineStyle="color:#6e6e6e" noWrap="true">
          <af:convertDateTime dateStyle="medium"/>
          </af:outputText>
        </af:panelGroupLayout>
      </af:panelGroupLayout>
    </dvt:timelineItem>
  </dvt:timelineSeries>
  <af:dragSource actions="COPY" discriminant="model"/>
  <af:dropTarget actions="COPY" dropListener="{dnd.handleDropOnTimeline}">
    <af:dataFlavor flavorClass="org.apache.myfaces.trinidad.model.RowKeySet"
      discriminant="model2"/>
  </af:dropTarget>
  <dvt:timeAxis id="ta1" scale="weeks"/>
  <dvt:timelineOverview id="ov1">
    <dvt:timeAxis id="ta2" scale="years"/>
  </dvt:timelineOverview>
</dvt:timeline>
<af:table var="row" value="{dnd.tableModel}" rowSelection="single"
  inlineStyle="width:370px;height:400px">
  <af:column headerText="ID" width="20">
    <af:outputText value="{row.id}"/>
  </af:column>
  <af:column headerText="Event" width="340">
    <af:outputText value="{row.description}"/>
  </af:column>
  <af:dragSource actions="COPY" discriminant="model2"/>
  <af:collectionDropTarget actions="COPY" modelName="model"
    dropListener="{dnd.handleDropOnTable}"/>
</af:table>
```

Using Different Output Modes

This chapter describes how to use ADF Faces to display pages in modes suitable for printing and emailing. Topics include how to print page contents using the `showPrintablePageBehavior` tag and how to create emailable pages with the request parameter `org.apache.myfaces.trinidad.agent.email=true`.

This chapter includes the following sections:

- [About Using Different Output Modes](#)
- [Displaying a Page for Print](#)
- [Creating Emailable Pages](#)

About Using Different Output Modes

The entire page of the web pages that comprise an ADF application may not be suitable for printing, but ADF supports rendering of the content so it is appropriate for print or email.

ADF Faces enables you to output your page in a simplified mode either for printing or for emailing. For example, you may want users to be able to print a page (or a portion of a page), but instead of printing the page exactly as it is rendered in a web browser, you want to remove items that are not needed on a printed page, such as scroll bars and buttons. If a page is to be emailed, the page must be simplified so that email clients can correctly display it.

Note:

By default, when the ADF Faces framework detects that an application is being crawled by a search engine, it outputs pages in a simplified format for the crawler, similar to that for an emailable page. If you want to generate special content for web crawlers, you can use the EL-reachable `Agent` interface to detect when an agent is crawling the site, and then direct the agent to a specified link, for example:

```
<c:if test="#{requestContext.agent.type == 'webcrawler'}">
  <af:link text="This Link is rendered only for web crawlers"
          destination="http://www.newPage.com"/>
</c:if>
```

For more information, see the Trinidad Javadoc.

For displaying printable pages, ADF Faces offers the `showPrintablePageBehavior` tag that, when used in conjunction with a command component, enables users to view a simplified version of a page in their browser, which they can then print.

For email support, ADF Faces provides an API that can be used to convert a page to one that is suitable for display in the Microsoft Outlook 2007, Mozilla Thunderbird 10.0.5, or Gmail email clients.

Tip:

The current output mode (email or printable) can be reached from `AdfFacesContext`. Because this context is EL-reachable, you can use EL to bind to the output mode from the JSP page. For example, you might allow a graphic to be rendered only if the current mode is not `email` using the following expression:

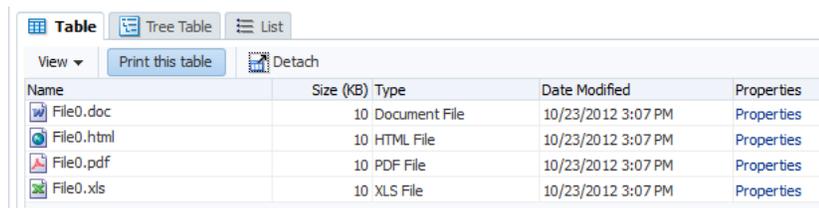
```
<af:activeImage source="/images/stockChart.gif"
rendered="#{adfFacesContext.outputMode != "email"}"/>
```

You can determine the current mode using `AdfFacesContext.getOutputMode()`.

Output Mode Use Cases

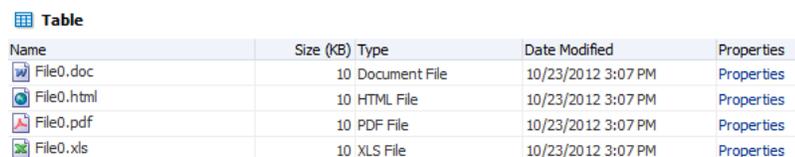
Most web pages are not suitable for print or for emailing, but users may need that functionality. For example, in the File Explorer application, you could place a `button` component inside the toolbar of the `panelCollection` component that contains a table, as shown in [Figure 40-1](#).

Figure 40-1 Button to Print Part of a Page



When the user clicks the button, the page is displayed in a new browser window (or tab, depending on the browser) in a simplified form, as shown in [Figure 40-2](#).

Figure 40-2 Printable Version of the Page



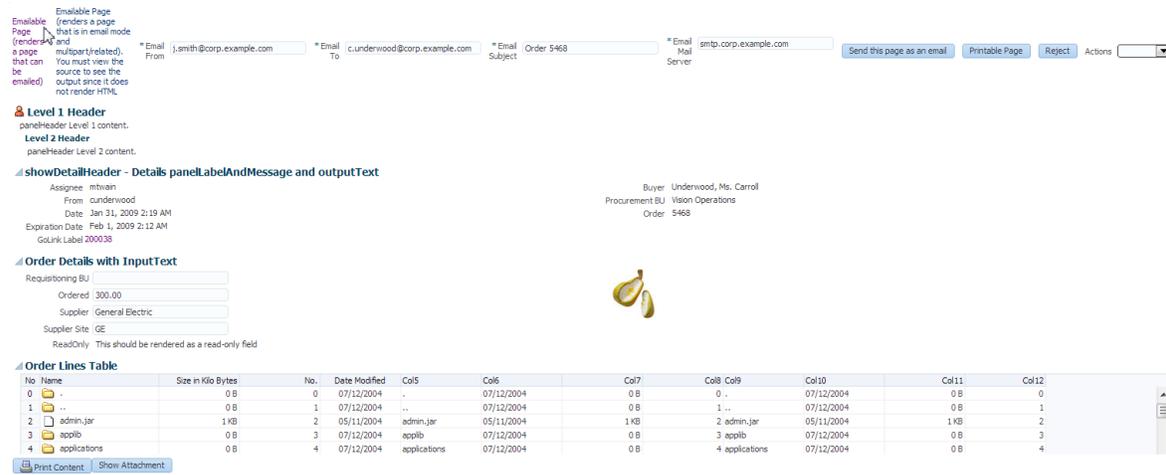
Name	Size (KB)	Type	Date Modified	Properties
File0.doc	10	Document File	10/23/2012 3:07 PM	Properties
File0.html	10	HTML File	10/23/2012 3:07 PM	Properties
File0.pdf	10	PDF File	10/23/2012 3:07 PM	Properties
File0.xls	10	XLS File	10/23/2012 3:07 PM	Properties

Only the contents of the table are displayed for printing. All extraneous components, such as the toolbar, buttons, and scroll bars, are not rendered.

There may be occasions when you need a page in your application to be emailed. For example, purchase orders created on the web are often emailed to the purchaser at the end of the session. However, because email clients do not support external stylesheets which are used to render to web browsers, you can't email the same page, as it would not be rendered correctly.

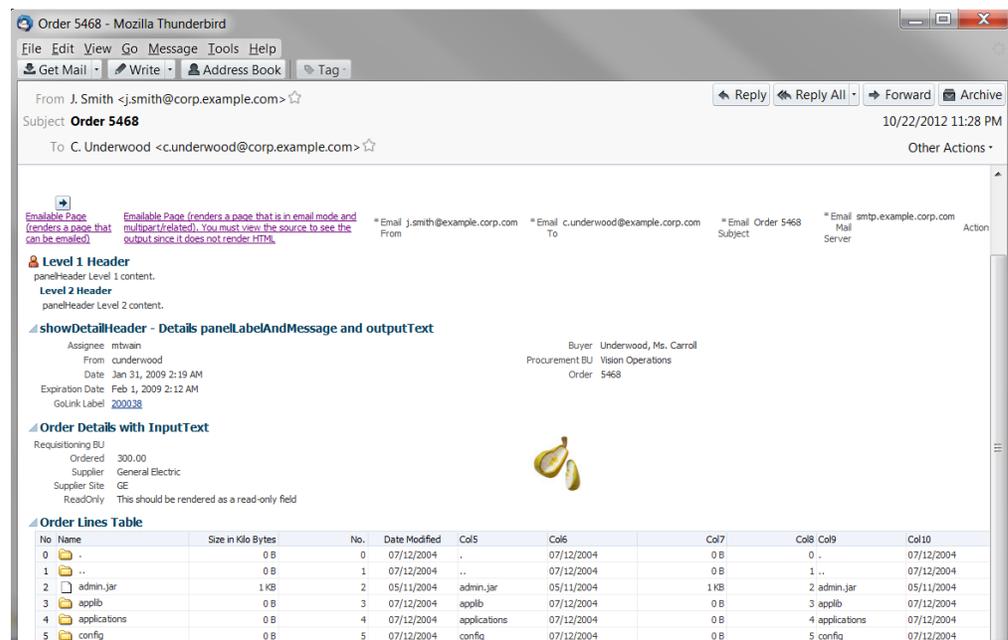
Say you have a page that displays a purchase order, as shown in [Figure 40-3](#).

Figure 40-3 Purchase Order Web Page



When the user clicks the [Emailable Page](#) link at the top, an `actionListener` method or another service appends `org.apache.myfaces.trinidad.agent.email=true` to the current URL and emails the page. [Figure 40-4](#) shows the page as it appears in an email client.

Figure 40-4 Page in an Email Client



Displaying a Page for Print

You can specify whether or not ADF Faces components that display in a web page should be printed when the ADF application user prints the web page.

The ADF Faces framework enables you to print versions of your pages that are suitable for printing when you place the `showPrintablePageBehavior` tag as a child to a command component. When clicked, the framework walks up the component tree, starting with the component that is the parent to the `printableBehavior` tag, until it reaches a `panelSplitter`, `panelAccordion`, or `popup` component, or the root of the tree (whichever comes first). The tree is rendered from there. Additionally, certain components that are not needed on a printed page (such as buttons and scrollbars) are omitted.

When the command component is clicked, the action event is canceled. Instead, a request is made to the server for the printable version of the page.

How to Use the `showPrintablePageBehavior` Tag

You use the `showPrintablePageBehavior` tag as a direct child of a command component.

Before you begin:

It may be helpful to have an understanding of how components will display in a printable page. For more information, see [Displaying a Page for Print](#).

To use the `showPrintablePageBehavior` tag:

1. In one of the layout components, add a command component in the facet that contains the content you would like to print. For procedures, see [How to Use Buttons and Links for Navigation and Deliver ActionEvents](#).

Note:

While you can insert a `showPrintablePageBehavior` component outside of a layout component to allow the user to print the entire page, the printed result will be roughly in line with the layout, which may mean that not all content will be visible. Therefore, if you want the user to be able to print the entire content of a facet, it is important to place the command component and the `showPrintablePageBehavior` component within the facet whose contents users would typically want to print. If more than one facet requires printing support, then insert one command component and `showPrintablePageBehavior` tag into each facet. To print all contents, the user then has to execute the print command one facet at a time.

2. In the Components window, from the Operations panel, drag a **Show Printable Page Behavior** and drop it as a child to the command component.

Creating Emailable Pages

Not all ADF Faces components that display in a web page of an ADF application can be rendered as output for email. However, ADF Faces supports proper rendering of a large number of components for a variety of email clients.

There may be occasions when you need a page in your application to be emailed. For example, purchase orders created on the web are often emailed to the purchaser at the end of the session. However, because email clients do not support external stylesheets which are used to render to web browsers, you can't email the same page, as it would not be rendered correctly.

The ADF Faces framework provides you with automatic conversion of a JSF page so that it will render correctly in the Microsoft Outlook 2007, Mozilla Thunderbird 10.0.5, or Gmail email clients.

Not all components can be rendered in an email client. The following components can be converted so that they can render properly in an email client:

- document
- panelHeader
- panelFormLayout
- panelGroupLayout
- panelList
- spacer
- showDetailHeader
- inputText (renders as readOnly)
- inputComboBoxListOfValues (renders as readOnly)
- inputNumberSlider (renders as readOnly)
- inputNumberSpinbox (renders as readOnly)
- inputRangeSlider (renders as readOnly)
- outputText
- selectOneChoice (renders as readOnly)
- panelLabelAndMessage
- image
- tree
- table
- treeTable
- column
- link (renders as text)

The emailable page will render all rows in a `table` and all nodes in a `treeTable` with each node expanded in the page, up to a limit. The default number of rows to display is 50. The maximum number of rows that a view object will retrieve is limited by the

data layer (through the `MaxFetchSize` property) or the application layer (through the `rowLimit` property specified in `adf-config.xml`). The `rowLimit` property is a global upper bound for all view object queries in the application. If `MaxFetchSize` is specified, `rowLimit` is ignored for that view object. You can also specify the maximum rows to display using the table attribute `rows` for emailable pages. The `rows` value will be used only if it is less than the defined value of `MaxFetchSize` or `rowLimit`. For tree and `treeTable`, use `nonScrollableRows` to limit the number of rows to display.

 **Note:**

If you include the `PanelGroupLayout` component while creating an emailable page, remove the `<td width="100%"></td>` entry within the `<table>... </table>` element in the HTML source. Otherwise, the footer row will not render properly in the email client.

By default, the emailable page will display with styles inlined directly on the relevant HTML element's style attribute rather than using an internal stylesheet. You can configure this behavior by setting the `web.xml` parameter `oracle.adf.view.DEFAULT_EMAIL_MODE` to either `internal` or `inline`.

How to Create an Emailable Page

You notify the ADF Faces framework to convert your page to be rendered in an email client by appending a request parameter to the URL for the page to be emailed.

Before you begin:

It may be helpful to have an understanding of how components will display in an emailable page. For more information, see [Creating Emailable Pages](#).

To create an emailable page:

1. Insert a command component onto the page to be emailed. For more information, see [Working with Navigation Components](#).
2. In a managed bean, create an `actionListener` method or another service that appends `org.apache.myfaces.trinidad.agent.email=true` to the current URL of the page to be emailed.

 **Note:**

By default, the framework will inline the style information onto each HTML element rather than creating an internal stylesheet. You can configure this behavior by setting the `web.xml` parameter `oracle.adf.view.DEFAULT_EMAIL_MODE` to either `internal` or `inline`. You can also force a request to render in either `internal` or `inline` mode by setting the request parameter to `org.apache.myfaces.trinidad.agent.email=internal` or `org.apache.myfaces.trinidad.agent.email=inline`.

3. Select the command component, and in the Properties window, set the method or service as the value for **ActionListener**.

 **Tip:**

If you want to be able to view the email offline, append the following request parameter to the URL of the page to be emailed:

```
org.apache.myfaces.trinidad.agent.email=true&oracle.adf.view.rich.rend
er.emailContentType=multipart/related
```

The framework will convert the HTML to MIME (multipart/related) and inline the images so the email can be viewed offline.

How to Test the Rendering of a Page in an Email Client

Before you complete the development of a page, you may want to test how the page will render in an email client. You can easily do this using a `Button` component.

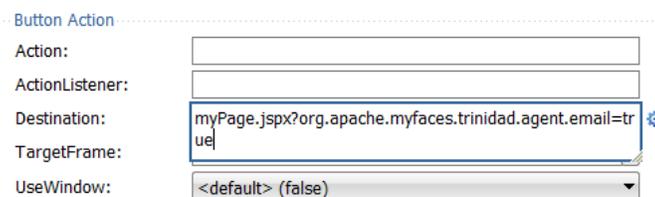
To test an emailable page:

1. In the Components window, from the General Controls panel, drag and drop a **Button** anywhere onto the page.
2. In the Design tab, right-click the Button and then in the Common section set the **Destination** to be the page's name plus
`org.apache.myfaces.trinidad.agent.email=true`.

For example, if your page's name is `myPage`, the value of the destination attribute should be:

```
myPage.jspx?org.apache.myfaces.trinidad.agent.email=true
```

Figure 40-5 Setting the Destination Attribute



Button Action

Action:

ActionListener:

Destination:

TargetFrame:

UseWindow:

3. Right-click the page and choose **Run** to run the page in the default browser.
The Create Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter must be at least eight characters and must have at least one non-alphabetic character.
4. Once the page displays in the browser, click the Button you added to the page. This will again display the page in the browser, but converted to a page that can be handled by an email client.
5. In your browser, view the source of the page. For example, in Mozilla Firefox, you would select **View > Page Source**. Select the entire source and copy it.
6. Create a new message in your email client. Paste the page source into the message and send it to yourself.

 **Tip:**

Because you are pasting HTML code, you will probably need to use an insert command to insert the HTML into the email body. For example, in Thunderbird, you would choose **Insert > HTML**.

7. If needed, create a skin specifically for the email version of the page using an agent. The following example shows how you might specify the border on a table rendered in email.

```
af|table {  
    border: 1px solid #636661;  
}  
  
@agent email {  
    af|table  
        {border:none}  
}  
  
af|table::column-resize-indicator {  
    border-right: 2px dashed #979991;  
}
```

For more information about creating skins, see [Customizing the Appearance Using Styles and Skins](#).

What Happens at Runtime: How ADF Faces Converts JSF Pages to Emailable Pages

When the ADF Faces framework receives the request parameter `org.apache.myfaces.trinidad.agent.email=true` in the Render Response phase, the associated phase listener sets an internal flag that notifies the framework to do the following:

- Remove any JavaScript from the HTML.
- Add all CSS to the page, but only for components included on the page and global styles (those without the pipe character (|) in the name).
- Remove the CSS link from the HTML.
- Convert all relative links to absolute links.
- Render images with absolute URLs.
- Inline the style information onto each HTML element rather than creating an internal stylesheet.

Additionally, if you add the parameter `oracle.adf.view.rich.render.emailContentType=multipart/related` the framework will convert the HTML to MIME (multipart/related) and inline the images so the email can be viewed offline. The full request parameter would be:

```
org.apache.myfaces.trinidad.agent.email=true&oracle.adf.view.rich.render.emailContent  
Type=multipart/related
```

41

Using the Active Data Service with an Asynchronous Backend

This chapter describes how to register an asynchronous backend with Active Data Service (ADS) to provide real-time data updates to ADF Faces components. This chapter includes the following sections:

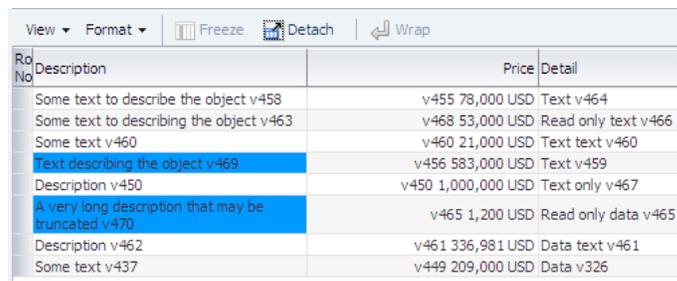
- [About the Active Data Service](#)
- [Process Overview for Using Active Data Service](#)
- [Implementing the ActiveModel Interface in a Managed Bean](#)
- [What You May Need to Know About Maintaining Read Consistency](#)
- [Passing the Event Into the Active Data Service](#)
- [Registering the Data Update Event Listener](#)
- [Configuring the ADF Component to Display Active Data](#)

About the Active Data Service

You can configure certain ADF Faces components to stream data from Active Data Service (ADS) to provide real-time data updates.

The Fusion technology stack includes the Active Data Service (ADS), which is a server-side push framework that allows you to provide real-time data updates for ADF Faces components. You bind ADF Faces components to a data source and ADS pushes the data updates to the browser client without requiring the browser client to explicitly request it. For example, you may have a table bound to attributes of an ADF data control whose values change on the server periodically, and you want the updated values to display in the table. You can create a Java bean to implement the `ActiveModel` interface and register it as an event listener to notify the component of a data event from the backend, and the component rerenders the changed data with the new value highlighted, as shown in [Figure 41-1](#).

Figure 41-1 Table Displays Updated Data as Highlighted



Row No	Description	Price	Detail
	Some text to describe the object v458	v455 78,000 USD	Text v464
	Some text to describing the object v463	v468 53,000 USD	Read only text v466
	Some text v460	v460 21,000 USD	Text text v460
	Text describing the object v469	v456 583,000 USD	Text v459
	Description v450	v450 1,000,000 USD	Text only v467
	A very long description that may be truncated v470	v465 1,200 USD	Read only data v465
	Description v462	v461 336,981 USD	Data text v461
	Some text v437	v449 209,000 USD	Data v326

Active Data Service Use Cases and Examples

Using ADS is an alternative to using automatic partial page rendering (PPR) to rerender data that changes on the backend as a result of business logic associated with the ADF data control bound to the ADF Faces component. Whereas automatic PPR requires sending a request to the server (typically initiated by the user), ADS enables changed data to be pushed from the data store as the data arrives on the server. Also, in contrast to PPR, ADS makes it possible for the component to rerender only the changed data instead of the entire component. This makes ADS ideal for situations where the application needs to react to data that changes periodically.

To use this functionality, you must configure the application to use ADS. If your application services do not support ADS, then you also need to create a proxy of the service so that the components can display the data as it updates in the source.

Any ADF Faces page can use ADS. However, you can configure only the following ADF Faces components and ADF Data Visualization (DVT) components to work with active data:

- `activeImage`
- `activeOutputText`
- `chart` (all types)
- `gauge` (all types)
- `pivotTable`
- `tree`
- `treeTable`
- `geoMap` (`mapPointTheme` only)
- `sunburst`
- `treemap`

For specific information about ADS support for DVT components, see [Active Data Support](#).

Additionally, note that collection-based components (such as `table`, `tree`, and `pivotTable`) support ADS only when the `outputText` component or `sparkChart` is configured to display the active data; other components are not supported inside the collection-based component.

For details about the active data service framework and important configuration information, see Using the Active Data Service in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Process Overview for Using Active Data Service

Active Data Service (ADS) in the ADF application is configured on the application level and on the ADF Faces component that you use to stream the active data.

To use ADS, you can optionally configure your application to determine the method of data transport, as well as other performance options.

Before you begin:

Complete the following tasks:

- Implement the logic to fire the active data events asynchronously from the data source. For example, this logic might be a business process that updates the database, or a JMS client that gets notified from JMS.
- The Active Data framework does not support complicated business logic or transformations that require the ADF runtime context, such as a user profile or security. For example, the framework cannot convert an ADF context locale-dependent value and return a locale-specific value. Instead, you need to have your data source handle this before publishing the data change event.
- Before users can run the ADF Faces page with ADS configured for the application, they must disable the popup blocker for their web browser. Active data is not supported in web browsers that have popup blockers enabled.

To use the Active Data Service:

1. Optionally, configure ADS to determine the data transport mode, as well as to set other configurations, such as a latency threshold and reconnect information. Configuration for ADS is done in the `adf-config.xml` file.

For details about configuring ADS, see *How to Configure the Active Data Service in Developing Fusion Web Applications with Oracle Application Development Framework*.

2. Optionally, configure a servlet parameter to specify the duration of the active session before it times out due to user inactivity. Configuration for the client-side servlet timeout parameter is done in the `web.xml` file.

For details about configuring the servlet timeout parameter, see *How to Configure Session Timeout for Active Data Service in Developing Fusion Web Applications with Oracle Application Development Framework*.

3. Create a backing bean that implements the `ActiveModel` interface and register it as the listener for active data events from your backend.
4. Create a class that extends the `BaseActiveDataModel` API to pass the `Event` object to the ADS framework.
5. Register a data change listener for data change events from the backend.
6. In the web page, configure the ADF Faces component to capture and display the pushed data by adding an expression to name the managed bean that implements the ADF component that you use to capture and display the pushed data.

Implementing the ActiveModel Interface in a Managed Bean

When you want to stream data to the ADF Faces components in the view layer of an ADF application, create a backing bean that implements the `ActiveModel` interface and register it as the listener for active data events from your backend.

Create a backing bean that contains the active model implementation as its property. This class uses an ADS decorator class to wrap the JSF model. This class should also implement a callback from the backend that will push data into the ADS framework.

You need to create a Java class that subclasses one of the following ADS decorator classes:

- `ActiveCollectionModelDecorator` class

- `ActiveGeoMapDataModelDecorator` class

These classes are wrapper classes that delegate the active data functionality to a default implementation of `ActiveDataModel`. The `ActiveDataModel` class listens for data change events and interacts with the Event Manager.

Specifically, when you implement the `ActiveModel` interface, you accomplish the following:

- Wraps the JSF model interface. For example, the `ActiveCollectionModelDecorator` class wraps the `CollectionModel` class.
- Generates active data events based on data change events from the data source.

To implement the `ActiveModel` interface, you need to implement methods on your Java class that gets the model to which the data is being sent and registers itself as the listener of the active data source (as illustrated in the following example):

1. Create a Java class that extends the decorator class appropriate for your component.

The following sample shows a `StockManager` class that extends `ActiveCollectionModelDecorator`. In this case, the data is displayed for an ADF Faces table component.

2. Implement the methods of the decorator class that will return the `ActiveDataModel` class and implement the method that returns the scalar model.

The following sample shows an implementation of the `getCollectionModel()` method that registers with an existing asynchronous backend. The method returns the list of stocks collection from the backend.

3. Implement a method that creates application-specific events that can be used to insert or update data on the active model.

The following example shows the `onStockUpdate()` callback method from the backend, which uses the active model (an instance of `ActiveStockModel`) to create `ActiveDataUpdateEvent` objects to push data to the ADF Faces component.

```
package sample.oracle.ads;

import java.util.List;
import sample.backend.IBackendListener;
import sample.bean.StockBean;
import sample.oracle.model.ActiveStockModel;

import oracle.adf.view.rich.event.ActiveDataEntry;
import oracle.adf.view.rich.event.ActiveDataUpdateEvent;
import oracle.adf.view.rich.model.ActiveCollectionModelDecorator;
import oracle.adf.view.rich.model.ActiveDataModel;

import oracle.adfinternal.view.faces.activedata.ActiveDataEventUtil;

import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.SortableModel;

// 1. This example wraps the existing collection model in the page and implements
//    the ActiveDataModel interface to enable ADS for the page.

public StockManager extends ActiveCollectionModelDecorator implements
                                IBackendListener
```

```

{
    // 2. Implement methods from ADF ActiveCollectionModelDecorator class to
    //     return the model.
    @Override
    public ActiveDataModel getActiveDataModel()
    {
        return stockModel;
    }

    @Override
    protected CollectionModel getCollectionModel()
    {
        if(collectionModel == null)
        {
            // connect to a backend system to get a Collection
            List<StockBean> stocks = FacesUtil.loadBackEnd().getStocks();
            // make the collection become a (Trinidad) CollectionModel
            collectionModel = new SortableModel(stocks);
        }

        return collectionModel;
    }

    // 3. Implement a callback method to create active data events and deliver to
    //     the ADS framework.

    /**
     * Callback from the backend to push new data to the decorator.
     * The decorator itself notifies the ADS system that there was a data change.
     *
     * @param key the rowKey of the updated Stock
     * @param updatedStock the updated stock object
     */
    @Override
    public void onStockUpdate(Integer rowKey, StockBean stock)
    {
        ActiveStockModel asm = getActiveStockModel();

        // start the preparation for the ADS update
        asm.prepareDataChange();

        // Create an ADS event, using an _internal_ util.
        // This class is not part of the API
        ActiveDataUpdateEvent event = ActiveDataEventUtil.buildActiveDataUpdateEvent(
            ActiveDataEntry.ChangeType.UPDATE, // type
            asm.getCurrentChangeCount(), // changeCount
            new Object[] {rowKey}, // rowKey
            null, //insertKey, null as we don't insert stuff
            new String[] {"value"}, // attribute/property name that changes
            new Object[] { stock.getValue()} // the payload for the above attribute
        );

        // Deliver the new Event object to the ADS framework
        asm.notifyDataChange(event);
    }

    /**
     * Typesafe caller for getActiveDataModel()
     * @return
     */
}

```

```

protected ActiveStockModel getActiveStockModel()
{
    return (ActiveStockModel) getActiveDataModel();
}

// properties
private CollectionModel collectionModel; // see getCollectionModel()...
private ActiveStockModel stockModel = new ActiveStockModel();
}

```

Register the class as a managed bean in the `faces-config.xml` file. The following example shows the bean `StockManager` is registered. Defining the managed bean allows you to specify the managed bean in an expression for the ADF Faces component's value property.

```

...
<managed-bean>
  <managed-bean-name>stockManager</managed-bean-name>
  <managed-bean-class>
    oracle.afdemo.view.feature.rich.StockManager
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

```

What You May Need to Know About Maintaining Read Consistency

Read consistency ensures that the ADF application pushes active data after the data refresh by the ADF Faces component but not before an active data event arrives at the web browser.

Using active data means that your component has two sources of data: the active data feed and the standard data fetch. Because of this, you must make sure your application maintains read consistency.

For example, say your page contains a table and that table has active data enabled. The table has two methods of delivery from which it updates its data: normal table data fetch and active data push. Say the back end data changes from `foo` to `bar` to `fred`. For each of these changes, an active data event is fired. If the table is refreshed *before* those events hit the browser, the table will display `fred` because standard data fetch will always get the latest data. But then, because the active data event might take longer, some time *after* the refresh the data change event would cause `foo` to arrive at the browser, and so the table would update to display `foo` instead of `fred` for a period of time. Therefore, you must implement a way to maintain the read consistency.

To achieve read consistency, the `ActiveDataModel` has the concept of a change count, which effectively timestamps the data. Both data fetch and active data push need to maintain this `changeCount` object by monotonically increasing the count, so that if any data returned has a lower `changeCount`, the active data event can throw it away. For information about how you can use your implementation of the `ActiveDataModel` class to maintain read consistency, see [Passing the Event Into the Active Data Service](#).

What You May Need to Know About Navigating Away From the ADS Enabled Page

When you configure the ADF application to push active data to ADF Faces components, you should handle navigation events that would terminate ADS so the user can optionally cancel the action.

When the user attempts to close the browser that displays an ADS-enabled page that they are viewing, ADS will stop on the browser native `window.unload` event. For those applications that want to allow the user to return to the page, the application needs to display a confirmation prompt so the user can cancel the action. In this case, the application should use the ADF client listener `beforeunload` and not the browser native `window.onbeforeunload` listener.

For an example of a client listener that can handle the ADF `beforeunload` event, see [How to Use an ADF Client Listener to Control Navigating Away From a JSF Page](#).

Passing the Event Into the Active Data Service

When you configure the ADF application to push active data to ADF Faces components, you create a class that extends `BaseActiveDataModel` class to pass the active data event created by your managed bean to the ADS framework.

You need to create a class that extends `BaseActiveDataModel` class to pass the event created by your managed bean. The `ActiveDataModel` class listens for data change events and interacts with the Event Manager. Specifically, the methods you implement do the following:

- Optionally, starts and stops the active data and the `ActiveDataModel` object, and registers and unregisters listeners to the data source.
- Manages listeners from the Event Manager and pushes active data events to the Event Manager.

The following examples shows the `notifyDataChange()` method of the model passes the `Event` object to the ADS framework, by placing the object into the `fireActiveDataUpdate()` method.

```
import java.util.Collection;

import java.util.concurrent.atomic.AtomicInteger;

import oracle.adf.view.rich.activeData.BaseActiveDataModel;
import oracle.adf.view.rich.event.ActiveDataUpdateEvent;

public class ActiveStockModel extends BaseActiveDataModel
{
    // ----- API from BaseActiveDataModel -----

    @Override
    protected void startActiveData(Collection<Object> rowKeys,
                                   int startChangeCount)
    {
        /* We don't do anything here as there is no need for it in this example.
         * You could use a listenerCount to see if the maximum allowed listeners
```

```
    * are already attached. You could register listeners here.
    */
    }

    @Override
    protected void stopActiveData(Collection<Object> rowKeys)
    {
        // same as above... no need to disconnect here
    }

    @Override
    public int getCurrentChangeCount()
    {
        return changeCounter.get();
    }

    // ----- Custom API -----

    /**
     * Increment the change counter.
     */
    public void prepareDataChange()
    {
        changeCounter.incrementAndGet();
    }

    /**
     * Deliver an ActiveDataUpdateEvent object to the ADS framework.
     *
     * @param event the ActiveDataUpdateEvent object
     */
    public void notifyDataChange(ActiveDataUpdateEvent event)
    {
        // Delegate to internal fireActiveDataUpdate() method.
        fireActiveDataUpdate(event);
    }

    // properties
    private final AtomicInteger changeCounter = new AtomicInteger();
}
```

Registering the Data Update Event Listener

When you configure the ADF application to push active data to ADF Faces components, you use the `faces-config.xml` file to register a data change listener for data changes on the backend.

You need to register a data change listener for data change events from the backend. The following example shows the listener bean `StockBackEndSystem` is registered in the `faces-config.xml` file. Note that for this example, expression language is used to inject a listener to the backend.

```
...
<managed-bean>
  <managed-bean-name>backend</managed-bean-name>
  <managed-bean-class>
    oracle.afdemo.backend.StockBackEndSystem
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
```

```

    <property-name>listener</property-name>
    <value>#{stockManager}</value>
  </managed-property>
</managed-bean>

```

Configuring the ADF Component to Display Active Data

When you configure the ADF application to push active data to ADF Faces components, you add an expression to name the managed bean that implements the ADF Faces component that you use to display the pushed data.

ADF components that display collection-based data can be configured to work with ADS and require no extra setup in the view layer. Once the listener is registered, you can use ADS to stream the data to the view layer. For example, imagine that your JSPX page uses a `table` component to display stock updates from a backend source on which you register a listener.

The following example shows the expression language used on the `table` component value attribute to receive the pushed data.

```

...
<f:view>
  <af:document id="d1">
    <af:form id="f1">
      <af:panelStretchLayout topHeight="50px" id="ps11">
        <f:facet name="top">
          <af:outputText value="Oracle ADF Faces goes Push!" id="ot1"/>
        </f:facet>
        <f:facet name="center">
          <!-- id="af_twocol_left_full_header_splitandstretched" -->
          <af:decorativeBox theme="dark" id="db2">
            <f:facet name="center">
              <af:panelSplitter orientation="horizontal"
                splitterPosition="100" id="ps1">
                <f:facet name="first">
                  <af:outputText value="Some content here." id="menu"/>
                </f:facet>
                <f:facet name="second">
                  <af:decorativeBox theme="medium" id="db1">
                    <f:facet name="center">
                      <af:table value="#{stockManager}" var="row"
                        rowBandingInterval="0"
                        id="table1" emptyText="No data...">
                        <af:column sortable="false" headerText="Name"
                          id="column1">
                          <af:outputText value="#{row.name}" id="outputText1"/>
                        </af:column>
                        <af:column sortable="false"
                          headerText="Value..." id="column2">
                          <af:outputText value="#{row.value}"
                            id="outputText2" />
                        </af:column>
                      </af:table>
                    </f:facet>
                  </af:decorativeBox>
                </f:facet>
              </af:panelSplitter>
            </f:facet>
          </af:decorativeBox>
        </f:facet>
      </af:panelStretchLayout>
    </af:form>
  </af:document>
</f:view>

```

```
        </af:panelStretchLayout>  
    </af:form>  
</af:document>  
</f:view>
```

Part VII

Appendices

This part contains reference information for your ADF Faces application, including configuration, message keys, shortcuts, designing and developing for mobile devices, layout themes, code samples, and troubleshooting.

Specifically, it contains the following appendices:

- [ADF Faces Configuration](#)
- [Message Keys for Converter and Validator Messages](#)
- [Keyboard Shortcuts](#)
- [Creating Web Applications for Touch Devices Using ADF Faces](#)
- [Quick Start Layout Themes](#)
- [Code Samples](#)
- [Troubleshooting ADF Faces](#)

A

ADF Faces Configuration

This appendix describes how to configure JSF and ADF Faces features in various XML configuration files, as well as how to retrieve ADF Faces configuration values using the RequestContext API and how to use JavaScript partitioning.

This appendix includes the following sections:

- [About Configuring ADF Faces](#)
- [Configuration in web.xml](#)
- [Configuration in faces-config.xml](#)
- [Configuration in adf-config.xml](#)
- [Configuration in adf-settings.xml](#)
- [Configuration in trinidad-config.xml](#)
- [Configuration in trinidad-skins.xml](#)
- [Using the RequestContext EL Implicit Object](#)
- [Performance Tuning](#)

About Configuring ADF Faces

ADF Faces applications use XML descriptor files to configure the ADF Faces and JSF features of the application. These configuration files identify the rules for navigating between pages, action listeners, validators, and many other things.

A JSF web application requires a specific set of configuration files, namely, `web.xml` and `faces-config.xml`. ADF Faces applications also store configuration information in the `adf-config.xml` and `adf-settings.xml` files. Because ADF Faces shares the same code base with MyFaces Trinidad, a JSF application that uses ADF Faces components for the UI also must include a `trinidad-config.xml` file, and optionally a `trinidad-skins.xml` file. For information about the relationship between Trinidad and ADF Faces, see [Introduction to ADF Faces](#).

Configuration in web.xml

The `web.xml` file acts as deployment descriptor for Java-based web applications. With the help of this file, you can control many aspects of the ADF Faces application's behavior, from preloading servlets, to restricting resource access, to controlling session time-outs.

Part of a JSF application's configuration is determined by the contents of its Java EE application deployment descriptor, `web.xml`. The `web.xml` file, which is located in the `/WEB-INF` directory, defines everything about your application that a server needs to know (except the root context path, which is automatically assigned for you in JDeveloper, or assigned by the system administrator when the application is deployed). Typical runtime settings in the `web.xml` file include initialization parameters and security settings.

The following is configured in the `web.xml` file for all applications that use ADF Faces:

- Context parameter `javax.faces.STATE_SAVING_METHOD` set to `client`
- MyFaces Trinidad filter and mapping
- MyFacesTrinidad resource servlet and mapping
- JSF servlet and mapping

 **Note:**

JDeveloper automatically adds the necessary ADF Faces configurations to the `web.xml` file for you the first time you use an ADF Faces component in an application.

For information about the required elements, see [What You May Need to Know About Required Elements in web.xml](#).

For information about optional configuration elements in `web.xml` related to ADF Faces, see [What You May Need to Know About ADF Faces Context Parameters in web.xml](#).

How to Configure for JSF and ADF Faces in web.xml

In JDeveloper, when you create a project that uses JSF technology, a starter `web.xml` file with default servlet and mapping elements is created for you in the `/WEB-INF` directory.

When you use ADF Faces components in a project (that is, a component tag is used on a page rather than just importing the library), in addition to default JSF configuration elements, JDeveloper also automatically adds the following to the `web.xml` file for you:

- Configuration elements that are related to MyFaces Trinidad filter and MyFaces Trinidad resource servlet
- Context parameter `javax.faces.STATE_SAVING_METHOD` with the value of `client`

When you elect to use JSF fragments in the application, JDeveloper automatically adds a JSP configuration element for recognizing and interpreting `.jsff` files in the application.

The following example shows the `web.xml` file with the default elements that JDeveloper adds for you when you use JSF and ADF Faces and `.jsff` files.

For information about the `web.xml` configuration elements needed for working with JSF and ADF Faces, see [What You May Need to Know About Required Elements in web.xml](#).

```
<?xml version = '1.0' encoding = 'windows-1252'?><web-app xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee">
  <description>Empty web.xml file for Web Application</description>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
```

```

    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>35</session-timeout>
</session-config>
<mime-mapping>
    <extension>html</extension>
    <mime-type>text/html</mime-type>
</mime-mapping>
<mime-mapping>
    <extension>txt</extension>
    <mime-type>text/plain</mime-type>
</mime-mapping>
</web-app>

```

 **Note:**

When you use ADF data controls to build databound web pages, the ADF binding filter and a servlet context parameter for the application binding container are added to the `web.xml` file.

Configuration options for ADF Faces are set in the `web.xml` file using `<context-param>` elements.

To add ADF Faces configuration elements in `web.xml`:

1. In the Applications window, double-click **web.xml** to open the file in the overview editor.

When you use the overview editor to add or edit entries declaratively, JDeveloper automatically updates the `web.xml` file for you.

2. To edit the XML code directly in the `web.xml` file, click **Source** at the bottom of the editor window.

When you edit elements in the XML editor, JDeveloper automatically reflects the changes in the overview editor.

For a list of context parameters you can add, see [What You May Need to Know About ADF Faces Context Parameters in web.xml](#).

What You May Need to Know About Required Elements in web.xml

The required, application-wide configuration elements for JSF and ADF Faces in the `web.xml` file are:

- Context parameter `javax.faces.STATE_SAVING_METHOD`: Specifies where to store the application's view state. By default this value is `client`, which stores the application's view state on the browser client. When set to `client`, ADF Faces then automatically uses token-based, client-side state saving. You can specify the number of tokens to use instead of using the default number of 15. For information

about state-saving context parameters, see [What You May Need to Know About ADF Faces Context Parameters in web.xml](#) .

- MyFaces Trinidad filter and mapping: Installs the MyFaces Trinidad filter `org.apache.myfaces.trinidad.webapp.TrinidadFilter`, which is a servlet filter that ensures ADF Faces is properly initialized, in part by establishing a `RequestContext` object. `TrinidadFilter` also processes file uploads. The filter mapping maps the JSF servlet's symbolic name to the MyFaces Trinidad filter. The forward and request dispatchers are needed for any other filter that is forwarding to the MyFaces Trinidad filter.

 **Tip:**

If you use multiple filters in your application, ensure that they are listed in the `web.xml` file in the order in which you want to run them. At runtime, the filters are called in the sequence listed in that file.

- MyFaces Trinidad resource servlet and mapping: Installs the MyFaces Trinidad resource servlet `org.apache.myfaces.trinidad.webapp.ResourceServlet`, which serves up web application resources (images, style sheets, JavaScript libraries) by delegating to a resource loader. The servlet mapping maps the MyFaces Trinidad resource servlet's symbolic name to the URL pattern. By default, JDeveloper uses `/adf/*` for MyFaces Trinidad Core, and `/afr/*` for ADF Faces.
- JSF servlet and mapping (added when creating a JSF page or using a template with ADF Faces components): The JSF servlet `servlet javax.faces.webapp.FacesServlet` manages the request processing lifecycle for web applications that utilize JSF to construct the user interface. The mapping maps the JSF servlet's symbolic name to the URL pattern, which can use either a path prefix or an extension suffix pattern.

By default JDeveloper uses the path prefix `/faces/*`, as shown in the following code:

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

For example, if your web page is `index.jspx`, this means that when the URL `http://localhost:8080/MyDemo/faces/index.jspx` is issued, the URL activates the JSF servlet, which strips off the `faces` prefix and loads the file `/MyDemo/index.jspx`.

What You May Need to Know About ADF Faces Context Parameters in web.xml

ADF Faces configuration options are defined in the `web.xml` file using `<context-param>` elements. For example:

```
<context-param>
  <param-name>oracle.adf.view.rich.LOGGER_LEVEL</param-name>
  <param-value>ALL</param-value>
</context-param>
```

You can also configure your application by setting the context parameter using Java API. See [What You May Need to Know About ADF Faces Window Manager Configuration](#).

The following context parameters are supported for ADF Faces.

State Saving

You can specify the following state-saving context parameters:

- `org.apache.myfaces.trinidad.CLIENT_STATE_METHOD`: Specifies the type of client-side state saving to use when client-side state saving is enabled by using `javax.faces.STATE_SAVING_METHOD`. The values for `CLIENT_STATE_METHOD` are:
 - `token`: (Default) Stores the page state in the session, but persists a token to the client. The simple token, which identifies a block of state stored back on the `HttpSession` object, is stored on the client. This enables ADF Faces to disambiguate the same page appearing multiple times. Failover is supported.
 - `all`: Stores all state information on the client in a (potentially large) hidden form field. It is useful for developers who do not want to use `HttpSession`.

Performance Tip:

Because of the potential size of storing all state information, you should set client-state saving to `token`.

- `org.apache.myfaces.trinidad.CLIENT_STATE_MAX_TOKENS`: Specifies how many tokens should be stored at any one time per user, when token-based client-side state saving is enabled. The default is 15. When the number of tokens is exceeded, the state is lost for the least recently viewed pages, which affects users who actively use the Back button or who have multiple windows opened at the same time. If you are building HTML applications that rely heavily on frames, you would want to increase this value.

Performance Tip:

In order to reduce live memory per session, consider reducing this value to 2. Reducing the state token cache to 2 means one Back button click is supported. For applications without support for a Back button, this value should be set to 1.

- `org.apache.myfaces.trinidad.COMPRESS_VIEW_STATE`: Specifies whether or not to globally compress state saving on the session. Each user session can have multiple `pageState` objects that heavily consume live memory and thereby impact performance. This overhead can become a much bigger issue in clustering when session replication occurs. The default is `false`.

 **Performance Tip:**

Latency can be reduced if the size of the data is compressed. To optimize performance, set to `true`.

Debugging

You can specify the following debugging context parameters:

- `org.apache.myfaces.trinidad.DEBUG_JAVASCRIPT`: ADF Faces, by default, obfuscates the JavaScript it delivers to the client, stripping comments and whitespace at the same time. This dramatically reduces the size of the ADF Faces JavaScript download, but it also makes it tricky to debug the JavaScript. Set to `true` to turn off the obfuscation during application development. Set to `false` for application deployment.
- `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION`: By default this parameter is `false`. If it is set to `true`, ADF Faces will automatically check the modification date of your JSF and CSS files, and discard the saved state when the files change.

 **Performance Tip:**

When set to `true`, this `CHECK_FILE_MODIFICATION` parameter adds overhead that should be avoided when your application is deployed. Set to `false` when deploying your application to a runtime environment.

- `oracle.adf.view.rich.LOGGER_LEVEL`: This parameter enables JavaScript logging when the default render kit is `oracle.adf.rich`. The default is `OFF`. If you wish to turn on JavaScript logging, use one of the following levels: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`, and `ALL`.

 **Performance Tip:**

JavaScript logging will affect performance. Set this value to `OFF` in a runtime environment.

- `oracle.adf.view.rich.REQUEST_ID_TRACING`: This parameter is used for diagnosing failed partial page rendering (PPR) requests by associating end user reports with corresponding entries in server-side logs. This is accomplished by appending the unique ECIF number for the server log to the PPR URL. By default this parameter is set to `off`. Set the parameter to `PPR` to activate the diagnostic functionality.

File Uploading

You can specify the following file upload context parameters:

- `org.apache.myfaces.trinidad.UPLOAD_MAX_MEMORY`: Specifies the maximum amount of memory that can be used in a single request to store uploaded files. The default is 100K.
- `org.apache.myfaces.trinidad.UPLOAD_MAX_DISK_SPACE`: Specifies the maximum amount of disk space that can be used in a single request to store uploaded files. The default is 2000K.
- `org.apache.myfaces.trinidad.UPLOAD_TEMP_DIR`: Specifies the directory where temporary files are to be stored during file uploading. The default is the user's temporary directory.

 **Note:**

The file upload initialization parameters are processed by the default `UploadedFileProcessor` only. If you replace the default processor with a custom `UploadedFileProcessor` implementation, the parameters are not processed.

Save Query Mode

You can specify the following parameter in `web.xml` to prevent users from creating a new search, updating an existing search, or deleting an existing saved search, when the user is in preview mode. This parameter also enables you to disable the saved searches when the user is in edit mode though the page composer is not running.

- `oracle.adf.faces.component.query.saveQueryMode`: specifies whether or not you want the saved searches to be displayed and used when the user is in preview mode or edit mode. Valid values are:
 - `default`: The user will be able to view and edit all saved searches.
 - `readonly`: The user will only be able to view and select saved searches. However, the user might not be able to update the saved searches.
 - `hidden`: The user will not be able to view any saved searches.

If you set the `saveQueryMode` attribute to a specific query component, it will override the global setting. See [How to Add the Query Component](#).

Resource Debug Mode

You can specify the following:

- `org.apache.myfaces.trinidad.resource.DEBUG`: Specifies whether or not resource debug mode is enabled. The default is `false`. Set to `true` if you want to enable resource debug mode. When enabled, ADF Faces sets HTTP response headers to let the browser know that resources (such as JavaScript libraries, images, and CSS) cannot be cached.

 **Tip:**

After turning on resource debug mode, clear your browser cache to force the browser to load the latest versions of the resources.

 **Performance Tip:**

In a production environment, this parameter should be removed or set to `false`.

User Customization

For information about enabling and using session change persistence, see [Allowing User Customization on JSF Pages](#).

Enabling the Application for Real User Experience Insight

Real User Experience Insight (RUEI) is a web-based utility to report on real-user traffic requested by, and generated from, your network. It measures the response times of pages and transactions at the most critical points in the network infrastructure. Session diagnostics allow you to perform root-cause analysis.

RUEI enables you to view server and network times based on the real-user experience, to monitor your Key Performance Indicators (KPIs) and Service Level Agreements (SLAs), and to trigger alert notifications on incidents that violate their defined targets. You can implement checks on page content, site errors, and the functional requirements of transactions. Using this information, you can verify your business and technical operations. You can also set custom alerts on the availability, throughput, and traffic of all items identified in RUEI.

Specify whether or not RUEI is enabled for `oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER` by adding the parameter to the `web.xml` file and setting the value to `true`. By default this parameter is not set or is set to `false`.

For information about RUEI, see *Enabling the Application for Real User Experience Insight and End User Monitoring* in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Assertions

You can specify whether or not assertions are used within ADF Faces using the `oracle.adf.view.rich.ASSERT_ENABLED` parameter. The default is `false`. Set to `true` to turn on assertions.

 **Performance Tip:**

Assertions will affect performance. Set this value to `false` in a runtime environment.

Dialog Prefix

To change the prefix for launching dialogs, set the `org.apache.myfaces.trinidad.DIALOG_NAVIGATION_PREFIX` parameter.

The default is `dialog:`, which is used in the beginning of the outcome of a JSF navigation rule that launches a dialog (for example, `dialog:error`).

Compression for CSS Class Names

You can set the `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION` parameter to determine compression of the CSS class names for skinning keys.

The default is `false`. Set to `true` if you want to disable the compression. The parameter also accepts EL expressions for its value.

 **Performance Tip:**

Compression will affect performance. In a production environment, set this parameter to `false`.

Control Caching When You Have Multiple ADF Skins in an Application

The skinning framework caches information in memory about the generated CSS file of each skin that an application requests. This could have performance implications if your application uses many different skins. Specify the maximum number of skins for which you want to cache information in memory as a value for the `org.apache.myfaces.trinidad.skin.MAX_SKINS_CACHED` parameter. The default value for this parameter is 20.

Test Automation

Use the `oracle.adf.view.rich.automation.ENABLED` parameter to notify ADF Faces that test automation is being used and turns on external component id attributes. See [Using Test Automation for ADF Faces](#).

UIViewRoot Caching

Use the `org.apache.myfaces.trinidad.CACHE_VIEW_ROOT` parameter to enable or disable `UIViewRoot` caching. When token client-side state saving is enabled, MyFaces Trinidad can apply an additional optimization by caching an entire `UIViewRoot` tree with each token. (Note that this does not affect thread safety or session failover.) This is a major optimization for AJAX-intensive systems, as postbacks can be processed far more rapidly without the need to reinstantiate the `UIViewRoot` tree.

You set the `org.apache.myfaces.trinidad.CACHE_VIEW_ROOT` parameter to `true` to enable caching. This is the default. Set the parameter to `false` to disable caching.

**Note:**

This type of caching is known to interfere with some other JSF technologies. In particular, the Apache MyFaces Tomahawk `saveState` component does not work, and template text in Facelets may appear in duplicate.

Themes and Tonal Styles

Although the `oracle.adf.view.rich.tonalstyles.ENABLED` parameter is still available for the purpose of backward compatibility, keep the parameter set to `false`, and use themes as a replacement style for the tonal style classes of `.AFDarkTone`, `.AFMediumTone`, `.AFLightTone` and `.AFDefaultTone`. Themes are easier to author than tonal styles; they rely on fewer selectors, and they avoid CSS containment selectors. For this reason they are less prone to bugs. Due to the limitation on the number of selectors in one CSS file, both tonal styles and themes cannot be supported in the same application.

Partial Page Rendering

Use the `org.apache.myfaces.trinidad.PPR_OPTIMIZATION` parameter to turn partial page rendering (PPR) optimization on and off. By default, this parameter is set to `off`. Set to `on` for improving the performance and efficiency of PPR.

Partial Page Navigation

Use the `oracle.adf.view.rich.pprNavigation.OPTIONS` parameter to turn partial page navigation on and off. By default, the value is `off`. Partial page navigation uses the same base page throughout the application, and simply replaces the body content of the page with each navigation. This processing results in better performance because JavaScript libraries and style sheets do not need to be reloaded with each new page. See [Using Partial Page Navigation](#).

Valid values are:

- `on`: PPR navigation is turned on for the application.

**Note:**

If you set the parameter to `on`, then you need to set the `partialSubmit` attribute to `true` for any command components involved in navigation. For information about `partialSubmit`, see [Events and Partial Page Rendering](#).

- `off`: PPR navigation is turned off for the application.
- `onWithForcePPR`: When an action on a command component results in navigation, the action will always be delivered using PPR, as if the component had

`partialSubmit` set to `true`. For information about `partialSubmit`, see [Events and Partial Page Rendering](#). If the component already has `partialSubmit` set to `true`, the framework does nothing. If `partialSubmit` is not set to `true`, the entire document is refreshed to ensure that old page refresh behavior is preserved. The entire document is also refreshed if the action component does not contain navigation.

Postback Payload Size Optimization

By default, during PPR, all fields are posted back to the server. For applications on high latency and/or low bandwidth networks, this could result in poor performance.

Use the `oracle.adf.view.rich.POSTBACK_PAYLOAD_TYPE` parameter to configure the application to only post back values of ADF Faces editable components when those values have changed since the last request.

Valid values are:

- `full`: all fields are posted (the default).
- `dirty`: only values of editable components that have changed since the last request are posted. However, the following will always be posted:
 - Values for components that have failed conversion or validation
 - Values for any third party components
 - Values for components that are bound to request scope or backing bean scope values.
 - Values for the `af:codeEditor`, `af:richTextEditor`, `af:inputFile` components

Note:

When you select `dirty`, client components are created for all ADF Faces editable components. This can result in slightly larger response payload sizes.

JavaScript Partitioning

Use the `oracle.adf.view.rich.libraryPartitioning.DISABLED` parameter to turn JavaScript partitioning on and off. By default, the value is `false` (enabled). JavaScript partitioning allows a page to download only the JavaScript needed by client components for that page.

Valid values are:

- `false`: JavaScript partitioning is enabled (the default).
- `true`: JavaScript partitioning is disabled.

For information about using and configuring JavaScript partitioning, see [JavaScript Library Partitioning](#).

Framebusting

Framebusting is a way to prevent clickjacking, which occurs when a malicious website pulls a page originating from another domain into a frame and overlays it with a counterfeit page, allowing only portions of the original, or clickjacked, page (for example, a button) to display. When users click the button, they in fact are clicking a button on the clickjacked page, causing unexpected results.

Say, for example, that your application is a web-based email application that resides in `DomainA`, and a website in `DomainB` clickjacks your page by creating a page with an `IFrame` that points to a page in your email application at `DomainA`. When the two pages are combined, the page from `DomainB` covers most of your page in the `IFrame`, and exposes only a button on your page that deletes all email for the account. Users, not realizing they are actually in the email application, may click the button and inadvertently delete all their email.

Framebusting prevents clickjacking by using the following JavaScript to block the application's pages from running in frames:

```
top.location.href = location.href;
```

The `org.apache.myfaces.trinidad.security.FRAME_BUSTING` context parameter manages framebusting in your application. Valid values are:

- `always`: The page will show an error and redirect whenever it attempts to run in a frame.
- `differentOrigin`: The page will show an error and redirect only when it attempts to run in a frame on a page that originates in a different domain (the default).
- `never`: The page can run in any frame on any originating domain.

Note:

For ADF Faces pages, this context parameter is ignored and will behave as if it were set to `never` when either of the following context parameters is set to `true`:

- `org.apache.myfaces.trinidad.util.ExternalContextUtils.isPortlet`
- `oracle.adf.view.rich.automation.ENABLED`

Because this is a MyFaces Trinidad parameter, it can also be used for those pages. Consult the MyFaces Trinidad documentation for information on using this parameter in a My Faces Trinidad application.

- `whitelist`: The page uses the value of the `oracle.adf.view.ALLOWED_ORIGINS` parameter to specify the origins that are allowed to frame documents in this application.

The `org.apache.myfaces.trinidad.security.FRAME_BUSTING` context parameter also supports `EL`. If `EL` is used, the `EL` will be evaluated once per user session.

If you configure your application to use framebusting by setting the `FRAME_BUSTING` parameter to `always`, then whenever a page tries to run in a frame, an alert is shown to

the user that the page is being redirected, the JavaScript code is run to define the page as topmost, and the page is disallowed to run in the frame.

If your application needs to use frames, you can set the parameter value to `differentOrigin`. This setting causes framebusting to occur only if the frame has the different origin as the parent page. This is the default setting.

 **Note:**

The origin of a page is defined using the domain name, application layer protocol, and in most browsers, TCP port of the HTML document running the script. Pages are considered to originate from the same domain if and only if all these values are exactly the same.

For example, these pages will fail the origin check due to the difference in port numbers:

- `http://www.example.com:8888/dir/page.html`
- `http://www.example.com:7777/dir/page.html`

For example, say you have a page named `DomainApage1` in your application that uses a frame to include the page `DomainApage2`. Say the external `DomainBpage1` tries to clickjack the page `DomainApage1`. The result would be the following window hierarchy:

- `DomainBpage1`
 - `DomainApage1`
 - * `DomainApage2`

If the application has framebusting set to be `differentOrigin`, then the framework walks the parent window hierarchy to determine whether any ancestor windows originate from a different domain. Because `DomainBpage1` originates from a different domain, the framebusting JavaScript code will run for the `DomainApage1` page, causing it to become the top-level window. And because `DomainApage2` originates from the same domain as `DomainApage1`, it will be allowed to run in the frame.

If you set the `FRAME_BUSTING` parameter to `whitelist`, the value of the `oracle.adf.view.ALLOWED_ORIGINS` parameter specifies the origins that are allowed to frame documents in this application.

The `ALLOWED_ORIGINS` parameter accepts a space-separated list. The format of the list is the Content Security Policy source list, as described in <http://www.w3.org/TR/CSP2/#source-list>. This space-separated list of origins allows wildcarding of the first segment of the host and port. For example:

```
http://*.oracle.com:*
```

Differences from the frame-ancestors specification are that the value `*` is supported and indicates that any origin may frame the content. If not specified or if an empty or null list returns from a value expression, the value defaults to `'self'`, indicating that only framing in the same origin is allowed.

When specifying the list of allowed origins, the current origin is not automatically allowed to frame the the application content. If you want the application to allow this, explicitly add `'self'` to the list of allowed origins.

The `ALLOWED_ORIGINS` parameter supports EL. The EL expression should return a collection of `String` with each collection item string representing one allowed origin. The EL expression is evaluated once per user session.

Version Number Information

Use the `oracle.adf.view.rich.versionString.HIDDEN` parameter to determine whether or not to display version information on a page's HTML. When the parameter is set to `false`, the HTML of an ADF Faces page contains information about the version of ADF Faces and other components used to create the page as shown in the following example.

```
</body><!--Created by Oracle ADF (ADF Faces API -  
11.1.1.4.0/ADF Faces Implementation - 11.1.1.4.0, RCF-revision: 39851 (branch:  
faces-1003-11.1.1.4.0, plugins: 1.2.3), Trinidad-revision: 1051544 (branch:  
1.2.12.3-branch, plugins: 1.2.10), build: adf-faces-rt_101221_0830, libNum:  
0355 powered by JavaServer Faces API 1.2 Sun Sep 26 03:21:43 EDT 2010  
(1.2)), accessibility (mode:null, contrast:standard, size:medium),  
skin:customSkin.desktop (CustomSkin)--></html>
```

Set the parameter to `true` to hide this version information. This is the default.

Note:

In a production environment, set this parameter to `true` to avoid security issues. It should be set to `false` only in a development environment for debugging purposes.

Suppressing Auto-Generated Component IDs

Use the `oracle.adf.view.rich.SUPPRESS_IDS` context parameter set to `auto` when programmatically adding an `af:outputText` or `af:outputFormatted` component as a partial target, that is, through a call to `addPartialTarget()`.

By default, this parameter is set to `explicit`, thereby reducing content size by suppressing both auto-generated and explicitly set component IDs except when either of the following is true:

- The component `partialTriggers` attribute is set
- The `clientComponent` attribute is set to `true`

In the case of a call to `addPartialTarget()`, the `partialTriggers` attribute is not set and the partial page render will not succeed. You can set the parameter to `auto` to suppress only auto-generated component IDs for these components.

ADF Faces Caching Filter

The ADF Faces Caching Filter (ACF) is a Java EE Servlet filter that can be used to accelerate web application performance by enabling the caching (and/or compression) of static application objects such as images, style sheets, and documents like `.pdf` and `.zip` files. These objects are cached in an external web cache such as Oracle Web Cache, Oracle Traffic Director, or in the browser cache. The cacheability of content is largely determined through URL-based rules defined by the web cache

administrator. Using ACF, the application administrator or author can define caching rules directly in the `adf-config.xml` file. For information about defining caching rules, see [Defining Caching Rules for ADF Faces Caching Filter](#).

ADF Faces tag library JARs include default caching rules for common resource types, such as `.js`, `.css`, and image file types. These fixed rules are defined in the `adf-settings.xml` file, and cannot be changed during or after application deployment. In the case of conflicting rules, caching rules defined by the application developer in `adf-config.xml` will take precedence. For information about settings in `adf-settings.xml`, see [What You May Need to Know About Elements in adf-settings.xml](#).

Oracle Web Cache and Oracle Traffic Director must be configured by the web cache administrator to route all traffic to the web application through the web cache. In the absence of the installation of Oracle Web Cache or Oracle Traffic Director, the caching rules defined in `adf-config.xml` will be applied for caching in the browser if the `<agent-caching>` child element is set to `true`. To configure the ACF to be in the URL request path, add the following servlet filter definitions in the `web.xml` file:

- **ACF filter class:** Specify the class to perform URL matching to rules defined in `adf-config.xml`
- **ACF filter mapping:** Define the URL patterns to match with the caching rules defined in `adf-config.xml`

The following example shows a sample ACF servlet definition.

```
<!-- Servlet Filter definition -->x
<filter>
  <filter-name>ACF</filter-name>
  <filter-class>oracle.adf.view.rich.webapp.AdfFacesCachingFilter</filter-class>
</filter>
<!-- servlet filter mapping definition -->
<filter-mapping>
  <filter-name>ACF</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

 **Note:**

The ACF servlet filter must be the first filter in the chain of filters defined for the application.

Configuring Native Browser Context Menus for Command Links

Use the `oracle.adf.view.rich.ACTION_LINK_BROWSER_CONTEXT_SUPPRESSION` context parameter to enable or disable the end user's browser to supply a context menu for ADF Faces command components that render a link. The context menu may present menu options that invoke a different action (for example, open a link in a new window) to that specified by the command component.

By default, this parameter is set to `yes`, thereby suppressing the rendering of a context menu for ADF Faces command components. By setting the parameter to `no`, you can disable this suppression and allow the native browser context menu to appear. For information about the ADF Faces command components for which you can configure this functionality, see

Internet Explorer Compatibility View Mode

Running ADF Faces applications in the compatibility mode of Microsoft Internet Explorer can cause unpredictable behavior. By default, when a user accesses an ADF Faces application and has their Internet Explorer browser set to compatibility mode, ADF Faces displays an alert asking the user to disable that mode.

Set the `oracle.adf.view.rich.HIDE_UNSUPPORTED_BROWSER_ALERTS` context parameter to `IECompatibilityModes` to hide these messages from the user.



Note:

Even when these messages are hidden, a warning-level log message is still reported to the JavaScript log, when the `oracle.adf.view.rich.LOGGER_LEVEL` parameter is set to `WARNING` or more verbose. See [Debugging](#).

Session Timeout Warning

When a request is sent to the server, a session timeout value is written to the page, based on the value of the `session-timeout` parameter. By default, this is set at 60 minutes when the `web.xml` file is created for you. See `session-config` in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

You can configure a session timeout warning interval using the context parameter `oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_TIMEOUT`. When set, the user is given the opportunity to extend the session in a warning dialog, and a notification is sent when the session has expired and the page is refreshed. To prevent the notification of the user too frequently, the timeout warning mechanism will try to determine that the client is still active by detecting the last keyboard or mouse event (not specific to the warning dialog). If the last activity was more recent than the session timeout warning interval, then a request is made to the server to reset the server session timeout value. Depending on the application security configuration, if no activity is detected, the user may be redirected to the log in page when the session expires.

If the value of `WARNING_BEFORE_TIMEOUT` is less than 120 seconds, if client state saving is used for the page, or if the session has been invalidated, the feature is disabled. The session timeout value is taken directly from the session.

The following example shows configuration of the warning dialog to display at 120 seconds before the timeout of the session.

```
<context-param>
  <param-name>oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_
    TIMEOUT</param-name>
  <param-value>120</param-value>
</context-param>
```

The default value of this parameter is 120 seconds. Additionally, to prevent notification of the user too frequently when the session time-out is set too short, the actual value of `WARNING_BEFORE_TIMEOUT` is determined dynamically, where the session time-out must be more than 2 minutes or the feature is disabled.

 **Note:**

The `WARNING_BEFORE_TIMEOUT` (expressed in seconds) should be set to less than the `session-timeout` (expressed in minutes)".

JSP Tag Execution in HTTP Streaming

Use the `oracle.adf.view.rich.tag.SKIP_EXECUTION` parameter to enable or disable JSP tag execution in HTTP streaming requests during the processing of JSP pages. Processing of facelets is not included.

By default, this parameter is set to `streaming`, where JSP tag execution is skipped during streaming requests. You can set the parameter to `off` to execute JSP tags per each request in cases where tag execution is needed by streaming requests.

Clean URLs

Historically, ADF Faces has used URL parameters to hold information, such as window IDs and state. However, URL parameters can prevent search engines from recognizing when URLs are actually the same, and therefore interfere with analytics. URL parameters can also interfere with bookmarking.

By default, ADF Faces removes these internally used dynamic URL parameters using the HTML5 History Management API. If that API is unavailable, then session cookies are used.

You can also manually configure how URL parameters are removed using the context parameter `oracle.adf.view.rich.prettyURL.OPTIONS`. Set the parameter to `off` so that no parameters are removed. Set the parameter to `useHistoryApi` to only use the HTML5 History Management API. If a browser does not support this API, then no parameters will be removed. Set the parameter to `useCookies` to use session cookies to remove parameters. If the browser does not support cookies, then no parameters will be removed.

Page Loading Splash Screen

Use the `oracle.adf.view.rich.SPLASH_SCREEN` parameter to enable or disable the splash screen that by default, displays as the page is loading, as shown in [Figure A-1](#).

Figure A-1 ADF Faces Splash Screen



By default, this parameter is set to `on`. You can set it to `off`, so that the splash screen will not display.

Graph and Gauge Image Format

Add the `oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT` parameter to change the default output format to HTML5 for graph and gauge components.

```
<context-param>
  <param-name>oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT</param-name>
  <param-value>HTML5</param-value>
</context-param>
```

By default, this parameter is added to all new applications. Valid values are `HTML5` (default) and `FLASH`.

Geometry Management for Layout and Table Components

Add the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter when you want to globally control how certain layout components and tables handle being stretched.

Whether or not certain layout components (`af:decorativeBox`, `af:panelAccordion`, `af:panelDashboard`, `af:panelStretchLayout`, `af:panelSplitter`, `af:panelTabbed`) can be stretched is based on the value of the `dimensionsFrom` attribute. The default setting for these components is `parent`, which means the size of the component is determined in the following order:

- From the `inlineStyle` attribute.
- If no value exists for `inlineStyle`, then the size is determined by the parent container (that is, the component will stretch).
- If the parent container is not configured or not able to stretch its children, the size will be determined by the skin.

However, if you always want these components to use `auto` as the value for the `dimensionsFrom` attribute (that is, the component stretches if the parent component allows stretching of its child, otherwise the size of the component is based on its child components), you can set the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter to `auto`. You can then use the `dimensionsFrom` attribute on an individual component to override this setting.

Similarly for tables, the `autoHeightRows` attribute determines whether or not the table will stretch. By default it is set to `-1`, which means the table size is based on the number of rows fetched. However, if the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter is set to `auto`, the table will stretch when the parent component allows stretching, and otherwise will be the number of rows determined by the table's `fetchSize` attribute.

By default, the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter is set to `legacy`, which means the components will use their standard default values.

Set `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter to `auto` when you want both layout components and tables to always stretch when the parent component allows stretching.

Rendering Tables Initially as Read Only

All the `clickToEdit` type tables are initially rendered as read only.

Setting the following `web.xml` param will force all `clickToEdit` type tables in the application to render as read only when the table is loaded initially. Clicking on a row will result in regular `clickToEdit` behavior. All tables using other `editingMode` types will be unaffected.

```
<context-param>
  <param-
name>oracle.adf.view.rich.table.clickToEdit.initialRender.readOnly</param-
name>
  <param-value>true</param-value>
</context-param>
```

Scrollbar Behavior in Tables

When you configure your table to use scrolling, in iOS operating systems, by default, the scrollbars only appear when you mouseover the content. Otherwise, they remain hidden. On other operating systems by default, the scrollbars always display. You can use the `oracle.adf.view.rich.table.scrollbarBehavior` context parameter to have all operating systems hide the scrollbars until a mouseover.

Figure A-2 shows a table whose scrollbars are hidden.

Figure A-2 Scrollbars are Hidden

Number	Name	Size of the file in Kilo B	Number	Date Modified	Col5	Col6
0	..	0 B	0	07/12/2004	.	07/12/2004
1	..	0 B	1	07/12/2004	..	07/12/2004
2	admin.jar	1 KB	2	05/11/2004	admin.jar	05/11/2004
3	applib	0 B	3	07/12/2004	applib	07/12/2004
4	applications	0 B	4	07/12/2004	applications	07/12/2004
5	config	0 B	5	07/12/2004	config	07/12/2004
6	connectors	0 B	6	07/12/2004	connectors	07/12/2004
7	database	0 B	7	07/12/2004	database	07/12/2004
8	default-web-app	0 B	8	07/12/2004	default-web-app	07/12/2004
9	iiop.jar	1,290 KB	9	05/11/2004	iiop.jar	05/11/2004
10	iiop_gen_bin.jar	37 KB	10	05/11/2004	iiop_gen_bin.jar	05/11/2004
11	iiop_rmic.jar	144 KB	11	05/11/2004	iiop_rmic.jar	05/11/2004
12	jazn	0 B	12	07/12/2004	jazn	07/12/2004
13	jazn.jar	266 KB	13	05/11/2004	jazn.jar	05/11/2004
14	jazncore.jar	553 KB	14	05/11/2004	jazncore.jar	05/11/2004
15	jaznplugin.jar	12 KB	15	05/11/2004	jaznplugin.jar	05/11/2004

Figure A-3 shows the same table when the user hovers over the table, and the scrollbars are shown.

Figure A-3 Scrollbars are Shown

Number	Name	Size of the file in Kilo	Number	Date Modified	Col5	Col6
0	.	0 B	0	07/12/2004	.	(▲)
1	..	0 B	1	07/12/2004	..	(☰)
2	admin.jar	1 KB	2	05/11/2004	admin.jar	()
3	applib	0 B	3	07/12/2004	applib	()
4	applications	0 B	4	07/12/2004	applications	()
5	config	0 B	5	07/12/2004	config	()
6	connectors	0 B	6	07/12/2004	connectors	()
7	database	0 B	7	07/12/2004	database	()
8	default-web-...	0 B	8	07/12/2004	default-web-app	()
9	iiop.jar	1,290 KB	9	05/11/2004	iiop.jar	()
10	iiop_gen_bin...	37 KB	10	05/11/2004	iiop_gen_bin.jar	()
11	iiop_rmic.jar	144 KB	11	05/11/2004	iiop_rmic.jar	()
12	jazn	0 B	12	07/12/2004	jazn	()
13	jazn.jar	266 KB	13	05/11/2004	jazn.jar	()
14	jazncore.jar	553 KB	14	05/11/2004	jazncore.jar	(▼)

Add the `oracle.adf.view.rich.table.scrollbarBehavior` when you want to globally control how the scrollbars are displayed. Set the value to `overlay` to have the scrollbars hidden until a mouseover occurs. Set the value to `default` to always show the scrollbars.

**Note:**

The `oracle.adf.view.rich.table.scrollbarBehavior` parameter also accepts EL expressions for its value.

Production Project Stage

Use the `javax.faces.PROJECT_STAGE` context parameter to:

- Force default values if not explicitly configured by XML, and
- Generate a warning if an incorrect value is set in a production environment.

By default the parameter is set to `PRODUCTION`. For example:

```
<param-value>PRODUCTION</param-value>
```

At runtime you can query the Application object for the configured value by calling `Application.getProjectStage()`. A warning is generated if one of the following parameters is set to `ON` in Production stage:

- `oracle.adf.view.rich.ASSERT_ENABLED`
- `oracle.adf.view.rich.automation.ENABLED` (whether set in web.xml or as a System property)

Additionally, Trinidad and ADF Faces parameters set to `FALSE` that generate the warning include:

- `org.apache.myfaces.trinidad.DEBUG_JAVASCRIPT`
- `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION`

- `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION`
- `org.apache.myfaces.trinidad.resource.DEBUG`
- `<debug-output>` (from `trinidad-config.xml` file)

 **Note:**

An additional warning is generated if the Trinidad parameters have been set through XML.

A warning is also generated if the `org.apache.myfaces.trinidad.CLIENT_ID_CACHING` System property has been disabled

Toggle Example Hints

The context parameter `oracle.adf.view.rich.component.DEFAULT_HINT_DISPLAY` is a center switch to turn off all the hints across the input fields when set to *none*.

The `web.xml` context parameter

`oracle.adf.view.rich.component.DEFAULT_HINT_DISPLAY` enables you to turn off hints across all components in the framework when set to *none*.

By default, the parameter is set to *auto* and allows the framework to decide the presentation mode for component hint. The default behavior is to show the hint as a note window tip on the component when it receives the focus.

The context parameter supports any EL expression returning `oracle.adf.view.rich.component.DefaultHintDisplay`.

```
<context-param>
  <param-name>oracle.adf.view.rich.component.DEFAULT_HINT_DISPLAY</param-name>
  <param-value>none</param-value>
</context-param>
```

What You May Need to Know About Other Context Parameters in web.xml

Other optional, application-wide context parameters are:

- `javax.faces.CONFIG_FILE`: Specifies paths to JSF application configuration resource files. Use a comma-separated list of application-context relative paths for the value, as shown in the following code. Set this parameter if you use more than one JSF configuration file in your application.

```
<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>
    /WEB-INF/faces-config1.xml,/WEB-INF/faces-config2.xml
  </param-value>
</context-param>
```

- `javax.faces.DEFAULT_SUFFIX`: Specifies a file extension (suffix) for JSP pages that contain JSF components. The default value is `.jsp`.

 **Note:**

This parameter value is ignored when you use prefix mapping for the JSF servlet (for example, `/faces`), which is done by default for you.

- `javax.faces.LIFECYCLE_ID`: Specifies a lifecycle identifier other than the default set by the `javax.faces.lifecycle.LifecycleFactory.DEFAULT_LIFECYCLE` constant.

 **Caution:**

Setting `LIFECYCLE_ID` to any other value will break ADF Faces.

Configuration in faces-config.xml

The `faces-config.xml` file contains the configurations for a web application built using JSF. This file allows you to configure the ADF Faces application, managed beans, converters, validators, and navigation.

The JSF configuration file is where you register a JSF application's resources such as custom validators and managed beans, and define all the page-to-page navigation rules. While an application can have any JSF configuration file name, typically the file name is the `faces-config.xml` file. Small applications usually have one `faces-config.xml` file.

When you use ADF Faces components in your application, JDeveloper automatically adds the necessary configuration elements for you into `faces-config.xml`. For information about the `faces-config.xml` file, see the Java EE 6 tutorial (<http://download.oracle.com/javasee/index.html>).

How to Configure for ADF Faces in faces-config.xml

In JDeveloper, when you create a project that uses JSF technology, an empty `faces-config.xml` file is created for you in the `/WEB-INF` directory. An empty `faces-config.xml` file is also automatically added for you when you create a new application workspace based on an application template that uses JSF technology (for example, the Java EE Web Application template). See [Creating an Application Workspace](#).

When you use ADF Faces components in your application, the ADF default render kit ID must be set to `oracle.adf.rich`. When you insert an ADF Faces component into a JSF page for the first time, or when you add the first JSF page to an application workspace that was created using the Fusion template, JDeveloper automatically inserts the default render kit for ADF components into the `faces-config.xml` file, as shown in the following example.

```
<?xml version="1.0" encoding="windows-1252"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee">
  <application>
```

```
<default-render-kit-id>oracle.adf.rich</default-render-kit-id>
</application>
</faces-config>
```

Typically, you would configure the following in the `faces-config.xml` file:

- Application resources such as message bundles and supported locales
- Page-to-page navigation rules
- Custom validators and converters
- Managed beans for holding and processing data, handling UI events, and performing business logic

 **Note:**

If your application uses ADF Controller, these items are configured in the `adfc-config.xml` file. See *Getting Started with ADF Task Flows in Developing Fusion Web Applications with Oracle Application Development Framework*.

In JDeveloper, you can use the declarative overview editor to modify the `faces-config.xml` file. If you are familiar with the JSF configuration elements, you can use the XML editor to edit the code directly.

To edit `faces-config.xml`:

1. In the Applications window, double-click **faces-config.xml** to open the file in the overview editor.

When you use the overview editor to add for example, managed beans and validators declaratively, JDeveloper automatically updates the `faces-config.xml` file for you.

2. To edit the XML code directly in the `faces-config.xml` file, click **Source** at the bottom of the editor window.

When you edit elements in the XML editor, JDeveloper automatically reflects the changes in the overview editor.

 **Tip:**

JSF allows more than one `<application>` element in a single `faces-config.xml` file. The Overview mode of the JSF Configuration Editor allows you to edit only the first `<application>` instance in the file. For any other `<application>` elements, you will need to edit the file directly using the XML editor.

Configuration in adf-config.xml

The adf-config.xml file contains application-level settings that manage the runtime infrastructure such as failover behavior for the application modules, caching of resource bundles, automated refresh of page bindings, and so on for your application.

The adf-config.xml file is used to configure application-wide features, like security, caching, and change persistence. Other Oracle components also configure properties in this file.

How to Configure ADF Faces in adf-config.xml

Before you can provide configuration for your application, you must first create the adf-config.xml file. Then you can add configuration for any application-wide Oracle ADF features that your application will use.

Before you begin:

If not already created, create a META-INF directory for your project.

To create and edit adf-config.xml:

1. If not already created, create a META-INF directory for your project.
2. In the Applications window, right-click the META-INF node, and choose **New > From Gallery**.
3. In the New Gallery, expand **General**, select **XML** and then **XML Document**, and click **OK**.

 **Tip:**

If you don't see the **General** node, click the **All Technologies** tab at the top of the Gallery.

4. In the Create XML File dialog, enter adf-config.xml as the file name and save it in the META-INF directory.
5. In the source editor, replace the generated code with the code shown in the following example.

```
<?xml version="1.0" encoding="utf-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:ads="http://xmlns.oracle.com/adf/activedata/config">

</adf-config>
```

6. You can now add the elements needed for the configuration of features you wish to use.

Defining Caching Rules for ADF Faces Caching Filter

Caching rules for the ADF Faces Caching Filter (ACF) are defined in the adf-config.xml file, located in the web-application's .adf/META-INF directory. You must

configure ACF to be in the request path for these URL matching rules. For information about adding the ACF servlet filter definition, see [ADF Faces Caching Filter](#).

The single root element for one or more caching rules is `<caching-rules>`, configured as a child of the `<adf-faces-config>` element in the namespace `http://xmlns.oracle.com/adf/faces/config`.

A `<caching-rule>` element defines each caching rule, evaluated in the order listed in the configuration file. The following example shows the syntax for defining caching rules in `adf-config.xml`.

```
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  <adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/config">
    <caching-rules xmlns="http://xmlns.oracle.com/adf/faces/rich/acf">
      <caching-rule id="cache-rule1">
        <cache>true|false</cache>
        <duration>3600</duration>
        <agent-caching>true|false</agent-caching>
        <agent-duration>4800</agent-duration>
        <compress>true|false</compress>
        <cache-key-pattern>...</cache-key-pattern>
        <search-key>
          <key>key1</key>
          <key>key2</key>
        </search-key>
        <varyBy>
          <vary-element>
            <vary-name><cookieName>|<headerName></vary-name>
            <vary-type>cookie|header</vary-type>
          </vary-element>
        </varyBy>
      </caching-rule>
    </caching-rules>
  </adf-faces-config>
</adf-config>
```

Each caching rule is defined in a `<caching-rule>` element. An optional `id` attribute can be defined to support rule location. [Table A-1](#) describes the `<caching-rule>` child elements used to define the parameters for caching or compressing the objects in the application.

Table A-1 AFC Caching Rule Elements and Attributes

Rule Element Children	Attribute Description and Value
<code><cache></code>	Specifies whether or not the object must be cached in the web cache. A value of <code>false</code> will ensure the object is never cached. The default is <code>true</code> .
<code><duration></code>	Defines the duration in seconds for which the object will be cached in the web cache. The default is 300 seconds.
<code><agent-caching></code>	Specify a value of <code>true</code> to use a browser cache in the absence of a web cache.
<code><agent-duration></code>	Defines the duration in seconds for which the object is cached in a browser cache. The default is <code>-1</code> . If <code><agent-caching></code> is <code>true</code> and <code><agent-duration></code> is not defined, then the value for <code><duration></code> is used instead.

Table A-1 (Cont.) AFC Caching Rule Elements and Attributes

Rule Element Children	Attribute Description and Value
<compress>	Specifies whether or not the object cached in the web cache must be compressed. The default value is <code>true</code> .
<cache-key-pattern>	Determines the URLs to match for the rule. One and only one <cache-key-pattern> element must be defined for the file extensions or the path prefix of a request URL. A <cache-key-pattern> value starting with a "*" value will be used as a file extension mapping, and others will be used as path prefix mapping.
<search-key> <key>	Defines the search keys tagged to the cached object. Each <search-key> can define one <search-key> element with one or more child <key> elements. The value of a search key is used in invalidating cached content. A default <search-key> is added at runtime for the context root of the application in order to identify all resources related to an application.
<varyBy> <vary-element> <vary-name> <vary-type>	<p>Used for versioning objects cached in the web cache. A <varyBy> element can have one or more <vary-element> elements that define the parameters for versioning a cached object. Most static resources will not require this definition.</p> <p>Each <vary-element> is defined by:</p> <ul style="list-style-type: none"> • <vary-name>: Valid values are <code>cookieName</code> for the name of the cookie whose value the response varies on, or <code>headerName</code> for the name of the HTTP header whose value determines the version of the object that is cached in the web cache. • <vary-type>: Valid values are <code>cookie</code> or <code>header</code>. <p>The web cache automatically versions request parameters. Multiple version of an object will be stored in web cache based on the request parameter.</p>

Configuring Flash as Component Output Format

By default, the application uses the output format specified for each component. For example, applications using ADF Data Visualization components can specify a Flash output format to display animation and interactivity effects in a web browser. If the component output format is Flash, and the user's platform doesn't support the Flash Player, as in Apple's iOS operating system, the output format is automatically downgraded to the best available fallback.

You can configure the use of Flash content across the entire application by setting a `flash-player-usage` context parameter in `adf-config.xml`. The valid settings include:

- `downgrade`: Specify that if the output format is Flash, but the Flash Player isn't available, then downgrade to the best available fallback. The user will not be prompted to download the Flash Player.
- `disabled`: Specify to disable the use of Flash across the application. All components will be rendered in their non-Flash versions, regardless of whether or not the Flash Player is available on the client.

The following example shows the syntax for application-wide disabling of Flash in `adf-config.xml`.

```
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  <adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/config">
    <flash-player-usage>disabled</flash-player-usage>
  </adf-faces-config></adf-config>
```

The context parameter also supports an EL Expression value. This allows applications to selectively enable or disable Flash for different parts of the application, or for different users, based on their preferences.

Using Content Delivery Networks

Content Delivery Networks (CDNs) improve web application performance by providing more efficient network access to content. Applications can use a variety of CDN configurations to optimize the user experience. An increasingly common configuration is to route all requests through a CDN. The CDN acts as a proxy between the client and the application. CDN-specific configuration tools can be used to specify caching and compression rules.

An alternate approach is to limit which requests are routed through the CDN. For example, only requests for auxiliary resources (images, JavaScript libraries, style sheets) might be directed to the CDN, while requests for application-generated HTML content can be served up directly. In this case, it is necessary to convert relative resource URIs to absolute URIs that point to the host that is serviced by the CDN.

For example, say your application-defined images are held in a local directory named `images`. Your code to reference images might look something like the following example:

```
<af:image source="/images/logo.png"
  shortDesc="My Company Logo"
  id="i1"/>
```

One way to indicate that the image should be retrieved from a CDN is to explicitly specify an absolute URI for the image source on the CDN, as shown in the following example:

```
<af:image source="http://mycdn.com/images/logo.png"
  shortDesc="My Company Logo"
  id="i1"/>
```

A downside of this approach is that it requires updating many locations (possibly every image reference) in the application, duplicating the CDN base URI across pages. This can make enabling and disabling CDN usage or switching from one CDN to another prohibitively difficult.

An alternative approach might be to EL bind resource-related attributes, as shown in the following example:

```
<af:image source="#{preferences.baseUri}/logo.png"
  shortDesc="My Company Logo"
  id="i1"/>
```

This approach allows the CDN base URI to be specified in a single location (for example, in a managed bean). However, it places a burden on application developers to use the correct EL expressions throughout their content.

Rather than repeating references to the CDN location (either directly or through EL expressions) throughout the application, ADF Faces provides a centralized mechanism for modifying resource URIs. This mechanism allows one or more prefixes, or "base resource URIs", to be specified for resources. These base resource URIs are defined in the application's `adf-config.xml` file, under the `http://xmlns.oracle.com/adf/rewrite/config` namespace.

For example, the following code sample specifies that all `png` images in the `images` directory should be rewritten to include the `http://mycdn.com` prefix.

```
<adf-uri-rewrite-config xmlns="http://xmlns.oracle.com/adf/rewrite/config">
  <resource-uris>
    <base-resource-uri uri="http://mycdn.com/">
      <match-pattern> ^/.*images/.*\.png$</match-pattern>
    </base-resource-uri>
  </resource-uris>
</adf-uri-rewrite-config>
```

The regular expression specified by the `<match-pattern>` element (`^/.*images/.*\.png$`) is tested against all resource URIs rendered by the application. Any matching URIs are transformed to include the prefix specified by the `<base-resource-uri>` element's URI attribute.

One advantage of this solution is that it can be used to modify not just application-defined resource URIs, but URIs for resources that are used by ADF Faces itself. To simplify this task, ADF Faces exposes a small set of aliases that can be used with the `<match-alias>` element in place of regular expressions.

For example, the configuration in the following example applies the `http://mycdn.com/` prefix to all images defined by ADF Faces components:

```
<adf-uri-rewrite-config xmlns="http://xmlns.oracle.com/adf/rewrite/config">
  <resource-uris>
    <base-resource-uri uri="http://mycdn.com/">
      <match-alias>af:images</match-alias>
    </base-resource-uri>
  </resource-uris>
</adf-uri-rewrite-config>
```

Unlike the regular expressions specified via `<match-pattern>` elements, the aliases used with `<match-alias>` do not match application-defined resources. So, for example, the `af:images` alias in the above configuration will cause images defined by ADF Faces components, such as the default background images and icons that come with ADF Faces, to be prefixed without also prefixing images that are explicitly bundled with the application.

In addition to the `af:images` alias, aliases are also provided for targeting the ADF Faces skins (style sheets), JavaScript libraries, and resource documents.

To set up URIs for a CDN:

1. Create or open the `adf-config.xml` file (see [How to Configure ADF Faces in `adf-config.xml`](#)).

2. In the overview editor, click the **View** navigation tab.
3. In the View page, in the Content Delivery Networks (CDN) section, click the **Add** icon to add a new row to the **Base Resource URIs** table.
4. In the new row, in the **URI** column, enter the URI for the CDN. This will be used to create the full URI for the given resource. In the **Secure URI** column, if needed, enter a prefix when the URI is secure, for example, "https://"

These entries create the `<base-resource-uri>` element.

5. If you want to share resources across multiple web applications, and each web application has its own context path, then the URIs to the shared copy of the resources cannot contain any application-specific segments.

If you want to remove the application-specific context path in the rewritten URI, select **Remove output context path during rewrite**.

6. Select any aliases that you want hosted by the CDN. The following aliases are available:
 - **af:coreScripts**: ADF Faces' JavaScript libraries (used in previous releases)
 - **af:documents**: ADF Faces' HTML resources (for example, `blank.html`)
 - **af:images**: ADF Faces' and Trinidad's image resources
 - **af:scripts**: ADF Faces' boot and core JavaScript libraries
 - **af:skins**: Skin-generated style sheets

These selections create the `<match-alias>` elements.

7. To have application-specific resources use the rewritten URI, create a pattern using a regular express for each resource type. Click the **Add** icon and enter the expression, for example:

```
^/.*\/images\/.*\.png$
```

Note:

All attribute values may be EL-bound. However, EL-bound attributes are only evaluated once (at parse time).

These entries create the `<match-pattern>` elements.

The expression will be tested against rendered resource URIs. If a match is found, the resource URI is prefixed with the URI specified by the base resource URI created in Step 4. Multiple patterns may be defined for each base resource URI.

Tip:

Note that in order to minimize runtime overhead, the results of resource URI rewriting are cached. To prevent excessive caching, pattern expressions should only target static resources. Dynamically generated, data-centric resources (for example, resources generated from unbounded query parameter values) must not be rewritten using the base resource URI mechanism.

The values specified in the match elements are compared against all URIs that pass through `ExternalContext.encodeResourceURL()`. If a URI matches, the prefix specified in the enclosing `<base-resource-uri>` element is applied.

The following example shows how an application might be configured to use a CDN.

```
<adf-uri-rewrite-config xmlns="http://xmlns.oracle.com/adf/rewrite/config">
  <resource-uris>
    <base-resource-uri uri="http://mycdn.com/"
                      secure-uri="https://mycdn.com"
                      output-context-path="remove">
      <match-pattern>^/.*\/images/.*\.\png$</match-pattern>
      <match-pattern>^/.*\.\png\?ln=images$</match-pattern>
      <match-alias>af:documents</match-alias>
      <match-alias>af:scripts</match-alias>
    </base-resource-uri>
  </resource-uris>
</adf-uri-rewrite-config>
```

What You May Need to Know About Skin Style Sheets and CDN

While you can use the `af:skins` alias to rewrite skin style sheets to point to the CDN, in cases where the CDN is configured to proxy requests back to the application server, problems can arise if the application is running in a clustered and/or load-balanced environment.

Skin style sheets are generated and stored on the server that rendered the containing page content. By routing the style sheet request through the CDN, server affinity may be lost (for example, if the CDN lives in a different domain, resulting in a loss of the session cookie). As a result, the style sheet request may be routed to a server that has not yet generated the requested style sheet. In such cases, the style sheet request will not complete successfully.

To avoid potential failures in load-balanced and/or clustered environments you should not rewrite skin style sheet URIs in cases where cookies or session affinity may be lost.

What You May Need to Know About Preparing Your Resource Files for CDNs

Skin style sheets and JavaScript partition files are dynamically generated at runtime. If you need to move these resources to the CDN's server, both MyFaces Trinidad and ADF Faces provide tools to pregenerate and save these files, so that they can then be uploaded to a static site.

To pregenerate skin style sheets, you use the Trinidad pregeneration service. For more information, see the Skinning chapter of the Trinidad developer's guide at <http://myfaces.apache.org/trinidad/>.

To pregenerate JavaScript partition files, you use the ADF Faces JavaScript library pregeneration service.

To use the ADF Faces JavaScript library pregeneration service:

1. Turn the service on using a system property.
 - Double-click the project. In the Project Properties dialog, select **Run/Debug** and click **Edit**.

- Select **Launch Settings**, and in the **Java Options** field, enter -
`oracle.adf.view.rich.libraryPartitioning.PREGENERATION_SERVICE=on.`

 **Note:**

When `oracle.adf.view.rich.libraryPartitioning.PREGENERATION_SERVICE` is set to `on`, all other (non-pregeneration) requests in the application will fail. Only set this to `on` when you will be pregenerating these files.

2. Set another system property to set a directory to hold the generated file. In the **Java Options** field, enter -
`oracle.adf.view.rich.libraryPartitioning.PREGENERATION_SERVICE_TARGET_DIRECTORY=/home/user/output`

If you do not set a directory, the files will be saved to the applications's temporary directory.

3. Send a request to the `/-adf-pregenerate-js-partitions` view id. This request can take the following optional parameters to constrain the files to be generated:
 - `accessibility`:
 - `screenReader`
 - `default` (this is the default)
 - `optimization`:
 - `none`
 - `simple` (this is the default)
 - `automation`:
 - `enabled`
 - `disabled` (this is the default)

For example, the following request generates both `screenReader` and `default` accessibility mode variants with simple JavaScript optimizations applied and automation disabled.

```
/root/faces/-adf-pregenerate-js-partitions?
accessibility=screenReader&accessibility=default
```

Configuration in `adf-settings.xml`

The `adf-settings.xml` file keeps the UI project configurations. This file is present in the `.adf\META-INF` directory of the ADF application.

The `adf-settings.xml` file holds project- and library-level settings such as ADF Faces help providers and caching/compression rules. The configuration settings for the `adf-settings.xml` files are fixed and cannot be changed during and after application deployment. There can be multiple `adf-settings.xml` files in an application. ADF settings file users are responsible for merging the contents of their configurations.

How to Configure for ADF Faces in `adf-settings.xml`

Before you can provide configuration for your application, you must first create the `adf-settings.xml` file. Then you can add the configuration for any project features that your application will use. For information about configurations in this file, see [What You May Need to Know About Elements in `adf-settings.xml`](#).

To create and edit `adf-settings.xml`:

1. The `adf-settings.xml` file must reside in a `META-INF` directory. Where you create this directory depends on how you plan on deploying the project that uses the `adf-settings.xml` file.
 - If you will be deploying the project with the application EAR file, create the `META-INF` directory in the `/application_name/.adf` directory.
 - If the project has a dependency on the `adf-settings.xml` file, and the project may be deployed separately from the application (for example a bounded task flow deployed in an ADF library), then create the `META-INF` directory in the `/src` directory of your view project.

Tip:

If your application uses Oracle ADF Model, then you can create the `META-INF` directory in the `/adfmsrc` directory.

2. In JDeveloper choose **File > New > From Gallery**.
3. In the New Gallery, expand **General**, select **XML** and then **XML Document**, and click **OK**.

Tip:

If you don't see the **General** node, click the **All Technologies** tab at the top of the Gallery.

4. In the source editor, replace the generated code with the code shown in the following example, using the correct settings for your web application root.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings"
              xmlns:wap="http://xmlns.oracle.com/adf/share/http/config" >
  <wap:adf-web-config xmlns="http://xmlns.oracle.com/adf/share/http/config">
    <web-app-root rootName="myroot" />
  </wap:adf-web-config>
</adf-settings>
```

5. You can now add the elements needed for the configuration of features you wish to use. See [What You May Need to Know About Elements in `adf-settings.xml`](#).
6. Save the file as `adf-settings.xml` to the `META-INF` directory created in Step 1.

What You May Need to Know About Elements in `adf-settings.xml`

The following configuration elements are supported in the `adf-settings.xml` file.

Help System

You register the help provider used by your help system using the following elements:

- `<adf-faces-config>`: A parent element that groups configurations specific to ADF Faces.
- `<prefix-characters>`: The provided prefix if the help provider is to supply help topics only for help topic IDs beginning with a certain prefix. This can be omitted if prefixes are not used.
- `<help-provider-class>`: The help provider class.
- `<custom-property>` and `<property-value>`: A property element that defines the parameters the help provider class accepts.

The following example shows an example of a registered help provider. In this case, there is only one help provider for the application, so there is no need to include a prefix.

```
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/settings">
  <help-provider prefix="MYAPP">
    <help-provider-class>
      oracle.adfdemo.view.webapp.MyHelpProvider
    </help-provider-class>
    <property>
      <property-name>myCustomProperty</property-name>
      <value>someValue</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

Caching Rules

Application-specific libraries and JARs contain a variety of resources that may require caching and/or compression of files. In the event of multiple libraries or JAR files, an application may include one or more `adf-setting.xml` files that contain various caching rules based on matching URLs. The caching rules are merged into an ordered list at runtime. If a request for a resource matches more than one caching rule, the rule encountered first in the list will be honored.

The ADF Faces JAR file includes default caching rules for common resource types, such as `.js`, `.css`, and image file types. These fixed rules are defined in the `adf-settings.xml` file, and cannot be changed during or after application deployment. Application developers can define application caching rules in the `adf-config.xml` file that take precedence over the rules defined in `adf-settings.xml`. The following example shows the `adf-settings.xml` file for the ADF Faces JAR.

```
<adf-settings>
  <adf-faces-settings>
    <caching-rules>
      <caching-rule id="cache css">
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.css</cache-key-pattern>
      </caching-rule>
      <caching-rule id="cache js">
```

```
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.js</cache-key-pattern>
    </caching-rule>
    <caching-rule id="cache png">
        <compress>false</compress>
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.png</cache-key-pattern>
    </caching-rule>
    <caching-rule id="cache jpg">
        <compress>false</compress>
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.jpg</cache-key-pattern>
    </caching-rule>
    <caching-rule id="cache jpeg">
        <compress>false</compress>
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.jpeg</cache-key-pattern>
    </caching-rule>
    <caching-rule id="cache gif">
        <compress>false</compress>
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.gif</cache-key-pattern>
    </caching-rule>
    <caching-rule id="cache html">
        <compress>true</compress>
        <duration>99999</duration>
        <agent-caching>true</agent-caching>
        <cache-key-pattern>*.html</cache-key-pattern>
    </caching-rule>
</caching-rules>
</adf-faces-settings>
</adf-settings>
```

Configuration in trinidad-config.xml

Apache MyFaces Trinidad forms the base for the ADF Faces component set. By default, the generated trinidad-config.xml file contains only the skin family name. However, trinidad-config.xml can be used to override the default configurations for accessibility settings, locale settings, state management, and so on.

When you create a JSF application using ADF Faces components, you configure ADF Faces features (such as skin family and level of page accessibility support) in the trinidad-config.xml file. Like faces-config.xml, the trinidad-config.xml file has a simple XML structure that enables you to define element properties using the JSF Expression Language (EL) or static values.

 **Note:**

You can also configure high availability testing support by setting a system property to use `org.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION`. See [What You May Need to Know About Configuring a System Property](#).

How to Configure ADF Faces Features in trinidad-config.xml

In JDeveloper, when you insert an ADF Faces component into a JSF page for the first time, a starter `trinidad-config.xml` file is automatically created for you in the `/WEB-INF` directory. The following example shows a starter `trinidad-config.xml` file.

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://xmlns.oracle.com/trinidad/config">

  <skin-family>alta</skin-family>
  <skin-version>v1</skin-version>

</trinidad-config>
```

By default, JDeveloper configures the `alta` skin family for a JSF application that uses ADF Faces. You can change this to `skyros`, `simple`, or use a custom skin. If you wish to use a custom skin, create the `trinidad-skins.xml` configuration file, and modify `trinidad-config.xml` file to use the custom skin. For information about creating custom skins, see [Customizing the Appearance Using Styles and Skins](#).

Typically, you would configure the following in the `trinidad-config.xml` file:

- Page animation
- Level of page accessibility support
- Time zone
- Enhanced debugging output
- Oracle Help for the Web (OHW) URL

You can also register a custom file upload processor for uploading files.

In JDeveloper, you can use the XML editor to modify the `trinidad-config.xml` file.

To edit `trinidad-config.xml`:

1. In the Applications window, double-click **trinidad-config.xml**.
2. In the XML editor, you can directly enter code into the XML editor, or you can use the Structure window to aid you in adding elements. To use the Structure window:
 - a. In the Structure window, right-click an element and choose either **Insert before** or **Insert after**, and then choose the element you wish to insert.
 - b. In the Structure window, double-click the newly inserted element. In the Properties window, enter a value or select one from a dropdown list (if available).

In most cases you can enter either a JSF EL expression (such as `#{view.locale.language=='en' ? 'minimal' : 'alta'}`) or a static value

(for example, `<debug-output>true</debug-output>`). EL expressions are dynamically reevaluated on each request, and must return an appropriate object (for example, a boolean object).

For a list of the configuration elements you can use, see [What You May Need to Know About Elements in trinidad-config.xml](#).

Once you have configured the `trinidad-config.xml` file, you can retrieve the property values programmatically or by using JSF EL expressions. See [Using the RequestContext EL Implicit Object](#).

What You May Need to Know About Elements in trinidad-config.xml

All `trinidad-config.xml` files must begin with a `<trinidad-config>` element in the `http://myfaces.apache.org/trinidad/config` XML namespace. The order of elements inside of `<trinidad-config>` does not matter. You can include multiple instances of any element.

Animation Enabled

Certain ADF Faces components use animation when rendering. For example, trees and tree tables use animation when expanding and collapsing nodes. The following components use animation when rendering:

- Table detail facet for disclosing and undisclosing the facet
- Trees and tree table when expanding and collapsing nodes
- Menus
- Popup selectors
- Dialogs
- Note windows and message displays

The type and time of animation used is configured as part of the skin for the application. See [Customizing the Appearance Using Styles and Skins](#).

You can set the `animation-enabled` element to either `true` or `false`, or you can use an EL expression that resolves to either `true` or `false`. By default `animation-enabled` is set to `true`.

Performance Tip:

While using animation can improve the user experience, it can increase the response time when an action is executed. If speed is the biggest concern, then set this parameter to `false`.

Skin Family

As described in [How to Configure ADF Faces Features in trinidad-config.xml](#), JDeveloper by default uses the `alta` skin family for a JSF application that uses ADF Faces. You can change the `<skin-family>` value to `skyros`, `simple`, or to a custom skin definition. For information about creating and using custom skins, see [Customizing the Appearance Using Styles and Skins](#).

You can use an EL expression for the skin family value, as shown in the following code:

```
<skin-family>#{prefs.proxy.skinFamily}</skin-family>
```

Time Zone and Year

To set the time zone used for processing and displaying dates, and the year offset that should be used for parsing years with only two digits, use the following elements:

- `<time-zone>`: By default, ADF Faces uses the time zone used by the application server if no value is set. If needed, you can use an EL expression that evaluates to a `TimeZone` object. This value is used by `org.apache.myfaces.trinidad.converter.DateTimeConverter` while converting strings to `Date`.
- `<two-digit-year-start>`: This value is specified as a Gregorian calendar year and is used by `org.apache.myfaces.trinidad.converter.DateTimeConverter` to determine 100 year range for creating dates from `String` with two digit years. The resulting `Date` will be within `two-digit-year-start` and `two-digit-year-start + 100`. This element defaults to the year 1950 if no value is set. If needed, you can use a static integer value, or an EL expression that evaluates to an `Integer` object.

For example, if no value is specified, the 100 year range defaults to [1950, 2050], and the date 01/01/10 is resolved to 01/01/2010.

Enhanced Debugging Output

By default, the `<debug-output>` element is `false`. ADF Faces enhances debugging output when you set `<debug-output>` to `true`. The following features are then added to debug output:

- Automatic indenting
- Comments identifying which component was responsible for a block of HTML
- Detection of unbalanced elements, repeated use of the same attribute in a single element, or other malformed markup problems
- Detection of common HTML errors (for example, `<form>` tags inside other `<form>` tags or `<tr>` or `<td>` tags used in invalid locations).

Performance Tip:

Debugging impacts performance. Set this parameter to `false` in a production environment.

Page Accessibility Level

Use `<accessibility-mode>` to define the level of accessibility support in an application. The supported values are:

- `default`: Output supports accessibility features.

- **inaccessible:** Accessibility-specific constructs are removed to optimize output size.

Language Reading Direction

By default, ADF Faces page rendering direction is based on the language being used by the browser. You can, however, explicitly set the default page rendering direction in the `<right-to-left>` element by using an EL expression that evaluates to a Boolean object, or by using `true` or `false`, as shown in the following code:

```
<!-- Render the page right-to-left for Arabic -->
<!-- and left-to-right for all other languages -->
<right-to-left>
  #{view.locale.language=='ar' ? 'true' : 'false'}
</right-to-left>
```

Currency Code and Separators for Number Groups and Decimal Points

To set the currency code to use for formatting currency fields, and define the separator to use for groups of numbers and the decimal point, use the following elements:

- **`<currency-code>`:** Defines the default ISO 4217 currency code used by the `org.apache.myfaces.trinidad.converter.NumberConverter` class to format currency fields that do not specify an explicit currency code in their own converter. Use a static value or an EL expression that evaluates to a `String` object. For example:

```
<!-- Set the currency code to US dollars. -->
<currency-code>USD</currency-code>
```

- **`<number-grouping-separator>`:** Defines the separator used for groups of numbers (for example, a comma). ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. You can use a static value or an EL expression that evaluates to a `Character` object. If set, this value is used by the `org.apache.myfaces.trinidad.converter.NumberConverter` class while parsing and formatting.

For example, to set the number grouping separator to a period when the German language is used in the application, use this code:

```
<!-- Set the number grouping separator to period for German -->
<!-- and comma for all other languages -->
<number-grouping-separator>
  #{view.locale.language=='de' ? '.' : ','}
</number-grouping-separator>
```

- **`<decimal-separator>`:** Defines the separator (for example, a period or a comma) used for the decimal point. ADF Faces automatically derives the separator from the current locale, but you can override this default by specifying a value in this element. You can use a static value or an EL expression that evaluates to a `Character` object. If set, this value is used by the `org.apache.mtfaces.trinidad.converter.NumberConverter` class while parsing and formatting.

For example, to set the decimal separator to a comma when the German language is used in the application, use this code:

```
<!-- Set the decimal separator to comma for German -->
<!-- and period for all other languages -->
<decimal-separator>
  #{view.locale.language=='de' ? ',' : '.'}
</decimal-separator>
```

Formatting Dates and Numbers Locale

By default, ADF Faces and MyFaces Trinidad will format dates (including the first day of the week) and numbers in the same locale used for localized text (which by default is the locale of the browser). If, however, you want dates and numbers formatted in a different locale, you can use the `<formatting-locale>` element, which takes an IANA-formatted locale (for example, `ja`, `fr-CA`) as its value. The contents of this element can also be an EL expression pointing at an IANA string or a `java.util.Locale` object.

Output Mode

To change the output mode ADF Faces uses, set the `<output-mode>` element, using one of these values:

- `default`: The default page output mode (usually `display`).
- `printable`: An output mode suitable for printable pages.
- `email`: An output mode suitable for emailing a page's content.

Number of Active PageFlowScope Instances

By default ADF Faces sets the maximum number of active `PageFlowScope` instances at any one time to 15. Use the `<page-flow-scope-lifetime>` element to change the number. Unlike other elements, you must use a static value: EL expressions are not supported.

File Uploading

While you can set file uploading parameters in `web.xml`, configuring file uploading parameters in `trinidad-config.xml` has the advantage of supporting EL Expressions that can be evaluated at runtime to change the value setting. The following elements are supported:

- `<uploaded-file-processor>`: This parameter must be the name of a class that implements the `org.apache.myfaces.trinidad.webapp.UploadedFileProcessor` interface, responsible for processing each individual uploaded file as it comes from the incoming request and making its contents available for the rest of the request. Most developers will find the default `UploadedFileProcessor` sufficient for their purposes, but applications that need to support uploading very large files may improve their performance by immediately storing files in their final destination, instead of requiring Apache Trinidad to handle temporary storage during the request.
- `<uploaded-file-max-memory>`: Used to set the maximum amount of memory used during the file upload process before the data will start writing out to disk. This setting directly overrides the `web.xml` setting `org.apache.myfaces.trinidad.UPLOAD_MAX_MEMORY`. This value can be hard coded or can be explicitly configured with an EL expression that returns a `Long` object.

- `<uploaded-file-max-disk-space>`: Used to set the maximum amount of disk space allowed for an uploaded file before an `EOFException` is thrown. This setting directly overrides the `web.xml` setting `org.apache.myfaces.trinidad.UPLOAD_MAX_DISK_SPACE`. This value can be hard coded or can be explicitly configured with an EL expression that returns a `Long` object.
- `<uploaded-file-max-disk-space>`: Used to change the default location uploaded files are stored. This setting directly overrides the `web.xml` setting `org.apache.myfaces.trinidad.UPLOAD_TEMP_DIR`. This value can be hard coded or can be explicitly configured with an EL expression that returns a `String` object.

Custom File Uploaded Processor

Most applications do not need to replace the default `UploadedFileProcessor` instance provided in ADF Faces, but if your application must support uploading of very large files, or if it relies heavily on file uploads, you may wish to replace the default processor with a custom `UploadedFileProcessor` implementation.

For example, you could improve performance by using an implementation that immediately stores files in their final destination, instead of requiring ADF Faces to handle temporary storage during the request. To replace the default processor, specify your custom implementation using the `<uploaded-file-processor>` element, as shown in the following code:

```
<uploaded-file-processor>  
  com.mycompany.faces.myUploadedFileProcessor  
</uploaded-file-processor>
```

Client-Side Validation and Conversion

ADF Faces validators and converters support client-side validation and conversion, as well as server-side validation and conversion. ADF Faces client-side validators and converters work the same way as the server-side validators and converters, except that JavaScript is used on the client.

The JavaScript-enabled validators and converters run on the client when the form is submitted; thus errors can be caught without a server roundtrip.

The `<client-validation-disabled>` configuration element is not supported in the rich client version of ADF Faces. This means you cannot turn off client-side validation and conversion in ADF Faces applications.

What You May Need to Know About Configuring a System Property

Some Trinidad configuration options are set by a system property. To support high availability testing, use `org.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION`. On the system property pass a comma-delimited set of case-insensitive values including:

- `NONE`: No state serialization checks are performed (the default).
- `ALL`: Perform all available tests (unless `NONE` is also specified, in which case `NONE` takes precedence).

- **SESSION:** Wrap the Session Map returned by the ExternalContext to test that only serializable objects are placed in the Session Map, throwing a CastCastException if the object is not serializable.
- **TREE:** Aggressively attempt to serialize the component state during state saving and throw an exception if serialization fails.
- **COMPONENT:** Aggressively attempt to serialize each component subtree's state during state saving in order to identify the problem component (slow).
- **PROPERTY:** Aggressively attempt to serialize each property value during state saving in order to identify the problem property (slow).

For example, the tester would initially start off validating if the session and JSF state is serializable by setting the system property to:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=session,tree
```

If a JSF state serialization is detected, the test is rerun with the component and property flags enabled as:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=all
```

Configuration in trinidad-skins.xml

Existing ADF Faces applications use the skin that the application was configured to use when the application was created. The `trinidad-config.xml` file helps you to create your own custom skin by extending one of the skins provided by ADF Faces.

By default, JDeveloper uses the `alta` skin family when you create JSF pages with ADF Faces components. The skin family is configured in the `trinidad-config.xml` file, as described in [How to Configure ADF Faces Features in trinidad-config.xml](#). If you wish to use a custom skin for your application, create a `trinidad-skins.xml` file, which is used to register custom skins in an application.

For information about creating custom skins, see [Customizing the Appearance Using Styles and Skins](#).

Using the RequestContext EL Implicit Object

All the properties defined in an `trinidad-config.xml` file can be retrieved programatically with the `org.apache.myfaces.trinidad.context.RequestContext` class. In addition, the `RequestContext` object also exposes several properties of type `java.util.Map` that help users write EL expressions.

In ADF Faces, you can use the EL implicit object `requestContext` to retrieve values from configuration properties defined in the `trinidad-config.xml` file. The `requestContext` implicit object, which is an instance of the `org.apache.myfaces.trinidad.context.RequestContext` class, exposes several properties of type `java.util.Map`, enabling you to use JSF EL expressions to retrieve context object property values.

For example, the EL expression `#{requestContext}` returns the `RequestContext` object itself, and the EL expression `#{requestContext.skinFamily}` returns the value of the `<skin-family>` element from the `trinidad-config.xml` file.

You can also use EL expressions to bind a component attribute value to a property of the `requestContext` implicit object. For example, in the EL expression that follows, the

`<currency-code>` property is bound to the `currencyCode` attribute value of the JSF `ConvertNumber` component:

```
<af:outputText>
  <f:convertNumber currencyCode="#{requestContext.currencyCode}"/>
</af:outputText>
```

You can use the following `requestContext` implicit object properties:

- `requestContext.accessibilityMode`: Returns the value of the `<accessibility-mode>` element from the `trinidad-config.xml` file.
- `requestContext.agent`: Returns an object that describes the client agent that is making the request and that is to display the rendered output. The properties in the agent object are:
 - `agentName`: Canonical name of the agent browser, (for example, `gecko` and `ie`).
 - `agentVersion`: Version number of the agent browser.
 - `capabilities`: Map of capability names (for example, `height`, `width`) and their values for the current client request.
 - `hardwareMakeModel`: Canonical name of the hardware make and model (for example, `nokia6600` and `sonyericssonP900`).
 - `platformName`: Canonical name of the platform (for example, `ppc`, `windows`, and `mac`).
 - `platformVersion`: Version number of the platform.
 - `type`: Agent type (for example, `desktop`, `pda`, and `phone`).
- `requestContext.clientValidationDisabled`: Returns the value of the `<client-validation-disabled>` element from the `trinidad-config.xml` file.
- `requestContext.colorPalette`: Returns a Map that takes color palette names as keys, and returns the color palette as a result. Each color palette is an array of `java.awt.Color` objects. Provides access to four standard color palettes:
 - `web216`: The 216 web-safe colors
 - `default49`: A 49-color palette, with one fully transparent entry
 - `opaque40`: A 49-color palette, without a fully transparent entry
 - `default80`: An 80-color palette, with one fully transparent entry
- `requestContext.currencyCode`: Returns the value of the `<currency-code>` element from the `trinidad-config.xml` file.
- `requestContext.debugOutput`: Returns the value of the `<debug-output>` element from the `trinidad-config.xml` file.
- `requestContext.decimalSeparator`: Returns the value of the `<decimal-separator>` element from the `trinidad-config.xml` file.
- `requestContext.formatter`: Returns a Map object that performs message formatting with a recursive Map structure. The first key must be the message formatting mask, and the second key is the first parameter into the message.
- `requestContext.helpSystem`: Returns a Map object that accepts help system properties as keys, and returns a URL as a result. For example, the EL expression

`#{requestContext.helpSystem['frontPage']}` returns a URL to the front page of the help system. This assumes you have configured the `<oracle-help-servlet-url>` element in the `trinidad-config.xml` file.

- `requestContext.helpTopic`: Returns a Map object that accepts topic names as keys, and returns a URL as a result. For example, the EL expression `#{requestContext.helpTopic['foo']}` returns a URL to the help topic "foo". This assumes you have configured the `<oracle-help-servlet-url>` element in the `trinidad-config.xml` file.
- `requestContext.numberGroupingSeparator`: Returns the value of the `<number-grouping-separator>` element from the `trinidad-config.xml` file.
- `requestContext.oracleHelpServletUrl`: Returns the value of the `<oracle-help-servlet-url>` element from the `trinidad-config.xml` file.
- `requestContext.outputMode`: Returns the value of the `<output-mode>` element from the `trinidad-config.xml` file.
- `requestContext.pageFlowScope`: Returns a map of objects in the `pageFlowScope` object.
- `requestContext.rightToLeft`: Returns the value of the `<right-to-left>` element from the `trinidad-config.xml` file.
- `requestContext.skinFamily`: Returns the value of the `<skin-family>` element from the `trinidad-config.xml` file.
- `requestContext.timeZone`: Returns the value of the `<time-zone>` element from the `trinidad-config.xml` file.
- `requestContext.twoDigitYearStart`: Returns the value of the `<two-digit-year-start>` element from the `trinidad-config.xml` file.

For a complete list of properties, refer to the *Java API Reference for Oracle ADF Faces* for `org.apache.myfaces.trinidad.context.RequestContext`.

 **Note:**

One instance of the `org.apache.myfaces.trinidad.context.RequestContext` class exists per request. The `RequestContext` class does not extend the JSF `FacesContext` class.

To retrieve a configuration property programmatically, first call the static `getCurrentInstance()` method to get an instance of the `RequestContext` object, and then call the method that retrieves the desired property, as shown in the following code:

```
RequestContext context = RequestContext.getCurrentInstance();

// Get the time-zone property
TimeZone zone = context.getTimeZone();

// Get the right-to-left property
if (context.isRightToLeft())
{
    .
    .
    .
}
```

Performance Tuning

Performance tuning is the process to identify and systematically eliminate configuration bottlenecks until the ADF Faces application meets its performance objectives.

In addition to the performance tips related to specific configuration options, see Oracle Application Development Framework Performance Tuning in *Tuning Performance*.

B

Message Keys for Converter and Validator Messages

This appendix lists all the message keys and message setter methods for ADF Faces converters and validators.

This chapter includes the following sections:

- [About ADF Faces Default Messages](#)
- [Message Keys and Setter Methods](#)
- [Converter and Validator Message Keys and Setter Methods](#)

About ADF Faces Default Messages

ADF Faces provides default text for messages that are displayed when validation or conversion fails. They work together to minimize the chance that a user would see an error message and maximize the chance that a user is able to enter valid values.

The `FacesMessage` class supports both summary and detailed messages. The convention is that:

- The summary message is defined for the main key. The key value is of the form `classname.MSG_KEY`.
- The detailed message is of the form `classname.MSG_KEY_detail`.

In summary, to override a detailed message you can either use the setter method on the appropriate class or enter a replacement message in a resource bundle using the required message key.

You can also override the message string globally instead of having to change the message string per instance. You use a message bundle so that the custom string will be available for all instances. For information about overriding default converter and validator error messages globally, see [How to Define Custom Validator and Converter Messages for All Instances of a Component](#).

Placeholders are used in detail messages to provide relevant details such as the value the user entered and the label of the component for which this is a message. The general order of placeholder identifiers is:

- component label
- input value (if present)
- minimum value (if present)
- maximum value (if present)
- pattern (if present)

You can also use message bundles to set message strings globally at the application level. See [How to Define Custom Validator and Converter Messages for All Instances of a Component](#).

Message Keys and Setter Methods

In ADF Faces, you can replace the default messages with your own messages by setting the text on the `xxxMessageDetail` attributes of the validator or converter or by binding those attributes to a resource bundle using an EL expression. You can also use placeholders in the messages to include relevant details.

The following information is given for each of the ADF Faces converter and validators:

- The set method you can use to override the message.
- The message key you can use to identify your own version of the message in a resource bundle.
- How placeholders can be used in the message to include details such as the input values and patterns.

Converter and Validator Message Keys and Setter Methods

A number of converters and validators provided by ADF Faces help you to add conversion and validation capabilities to ADF Faces input components in your application.

The following subsections give the reference details for all ADF Faces converter and validator detail messages.

af:convertColor

Converts strings representing color values to and from `java.awt.Color` objects. The set of patterns used for conversion can be overridden.

Convert color: Input value cannot be converted to a color based on the patterns set

Set method:

```
setMessageDetailConvert(java.lang.String convertBothMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.ColorConverter.CONVERT_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} A color example

af:convertDateTime

Converts a string to and from `java.util.Date`, and the converse based on the pattern and style set.

Convert date and time: Date-time value that cannot be converted to Date object when type is set to both

Set method:

```
setMessageDetailConvertBoth(java.lang.String convertBothMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.DateTimeConverter.CONVERT_BOTH_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} Example of the format the converter is expecting

Convert date: Input value cannot be converted to a Date when the pattern or secondary pattern is set or when type is set to date

Set method:

```
setMessageDetailConvertDate(java.lang.String convertDateMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.DateTimeConverter.CONVERT_DATE_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} Example of the format the converter is expecting

Convert date: Input value cannot be converted to a Date when the pattern or secondary pattern is set or when type is set to time

Set method:

```
setMessageDetailConvertTime(java.lang.String convertTimeMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.DateTimeConverter.CONVERT_TIME_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} Example of the format the converter is expecting

af:convertNumber

Provides an extension of the standard JSF `javax.faces.convert.NumberConverter` class. The converter provides all the standard functionality of the default `NumberConverter` and is strict while converting to an object.

Convert number: Input value cannot be converted to a Number, based on the pattern set

Set method:

```
setMessageDetailConvertPattern(java.lang.String convertPatternMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.NumberConverter.CONVERT_PATTERN_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The specified conversion pattern

Convert number: Input value cannot be converted to a Number when type is set to number and pattern is null or not set

Set method:

```
setMessageDetailConvertNumber(java.lang.String convertNumberMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.NumberConverter.CONVERT_NUMBER_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user

Convert number: Input value cannot be converted to a Number when type is set to currency and pattern is null or not set

Set method:

```
setMessageDetailConvertCurrency(java.lang.String convertCurrencyMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.NumberConverter.CONVERT_CURRENCY_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user

Convert number: Input value cannot be converted to a Number when type is set to percent and pattern is null or not set

Set method:

```
setMessageDetailConvertPercent(java.lang.String convertPercentMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.convert.NumberConverter.CONVERT_PERCENT_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user

af:validateByteLength

Validates the byte length of strings when encoded.

Validate byte length: The input value exceeds the maximum byte length

Set method:

```
setMessageDetailMaximum(java.lang.String maximumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.ByteLengthValidator.MAXIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} Maximum length

af:validateDateRestriction

Validates that the date is valid with some given restrictions.

Validate date restriction - Invalid Date: The input value is invalid when invalidDate is within the list of invalidDays

Set method:

```
setMessageDetailInvalidDays(java.lang.String invalidDays)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateRestrictionValidator.WEEKDAY_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The invalid date

Validate date restriction - Invalid day of the week: The input value is invalid when the value is one of the weekdays specified in invalidDaysOfWeek

Set method:

```
setMessageDetailInvalidDaysOfWeek(java.lang.String invalidDaysOfWeek)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateRestrictionValidator.DAY_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The invalid day of week

Validate date restriction - Invalid month: The input value is invalid when the value has a month specified in `invalidMonths`

Set method:

```
setMessageDetailInvalidMonths(java.lang.String invalidMonths)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateRestrictionValidator.MONTH_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The invalid month

af:validateDateTimeRange

Validates that the date entered is within a given range.

Validate date-time range: The input value exceeds the `maximum` value set

Set method:

```
setMessageDetailMaximum(java.lang.String maximumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateTimeRangeValidator.MAXIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The maximum allowed date

Validate date-time range: The input value is less than the `minimum` value set

Set method:

```
setMessageDetailMinimum(java.lang.String minimumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateTimeRangeValidator.MINIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed date

Validate date-time range: The input value is not within the range, when `minimum` and `maximum` are set

Set method:

```
setMessageDetailNotInRange(java.lang.String notInRangeMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DateTimeRangeValidator.NOT_IN_RANGE_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed date
- {3} The maximum allowed date

af:validateDoubleRange

Validates that the value entered is within a given range.

Validate double range: The input value exceeds the `maximum` value set

Set method:

```
setMessageDetailMaximum(java.lang.String maximumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DoubleRangeValidator.MAXIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The maximum allowed value

Validate double range: The input value is less than the `minimum` value set

Set method:

```
setMessageDetailMinimum(java.lang.String minimumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DoubleRangeValidator.MINIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed value

Validate double range: The input value is not within the range, when `minimum` and `maximum` are set

Set method:

```
setMessageDetailNotInRange(java.lang.String notInRangeMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.DoubleRangeValidator.NOT_IN_RANGE_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed value
- {3} The maximum allowed value

af:validateLength

Validates that the value entered is within a given range.

Validate length: The input value exceeds the maximum value set

Set method:

```
setMessageDetailMaximum(java.lang.String maximumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.LengthValidator.MAXIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The maximum allowed length

Validate length: The input value is less than the minimum value set

Set method:

```
setMessageDetailMinimum(java.lang.String minimumMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.LengthValidator.MINIMUM_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed length

Validate length: The input value is not within the range, when minimum and maximum are set

Set method:

```
setMessageDetailNotInRange(java.lang.String notInRangeMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.LengthValidator.NOT_IN_RANGE_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The minimum allowed length
- {3} The maximum allowed length

af:validateRegExp

Validates an expression using Java regular expression syntax.

Validate regular expression: The input value does not match the specified pattern

Set method:

```
setMessageDetailNoMatch(java.lang.String noMatchMessageDetail)
```

Message key:

```
org.apache.myfaces.trinidad.validator.RegExpValidator.NO_MATCH_detail
```

Placeholders:

- {0} The label that identifies the component
- {1} Value entered by the user
- {2} The expected pattern

C

Keyboard Shortcuts

This appendix describes the keyboard shortcuts that can be used instead of pointing devices.

This appendix includes the following sections:

- [About Keyboard Shortcuts](#)
- [Tab Traversal](#)
- [Shortcut Keys](#)
- [Default Cursor or Focus Placement](#)
- [The Enter Key](#)

About Keyboard Shortcuts

Keyboard shortcuts are helpful to users as they act as an alternative to mouse. Using keyboard shortcuts for ADF Faces applications can greatly increase your productivity, reduce repetitive strain, and help keep you focused.

Keyboard shortcuts provide an alternative to pointing devices for navigating the page. There are five types of keyboard shortcuts that can be provided in ADF Faces applications:

- Tab traversal, using Tab and Shift+Tab keys: Moves the focus through UI elements on a screen.
- Accelerator keys (*hot keys*): bypasses menu and page navigation, and performs an action directly, for example, Ctrl+C for Copy.
- Access keys: Moves the focus to a specific UI element, for example, Alt+F for the File menu.
- Default cursor/focus placement: Puts the initial focus on a component so that keyboard users can start interacting with the page without excessive navigation.
- Enter key: Triggers an action when the cursor is in certain fields or when the focus is on a link or button.

Keyboard shortcuts are not required for accessibility. Users should be able to navigate to all parts and functions of the application using the Tab and arrow keys, without using any keyboard shortcuts. Keyboard shortcuts merely provide an additional way to access a function quickly.

It is the application developer's responsibility to provide user assistance that identifies the application's available keyboard shortcuts. If the application includes an ADF component that provides keyboard shortcuts, the developer should also provide a popup or other help that details the keyboard shortcuts available for the ADF component.

Tab Traversal

Tab Traversal can be defined as an order in which the elements of the ADF Faces user interface receive keyboard focus on successive passes of the Tab key.

Tab traversal allows the user to move the focus through different UI elements on a page.

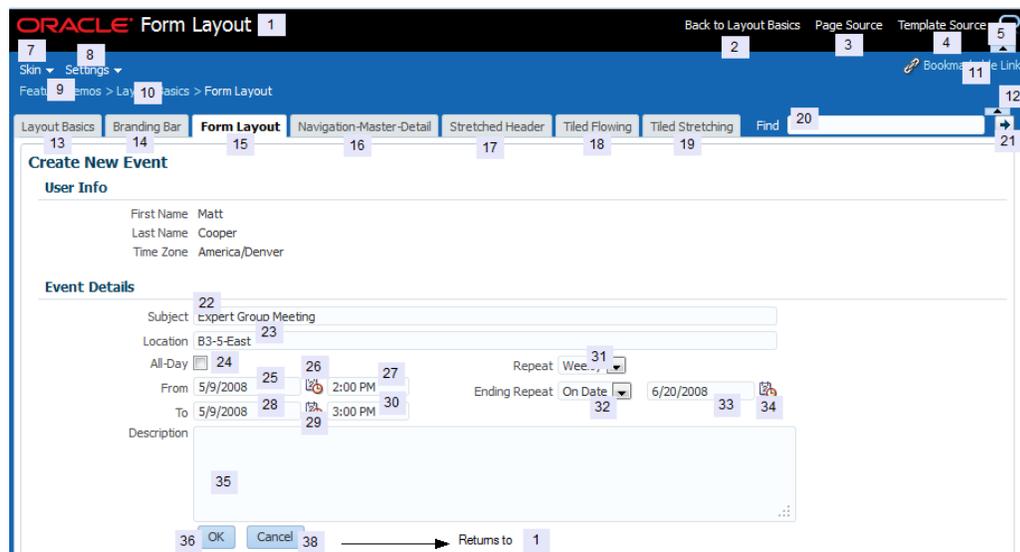
All active elements of the page are accessible by Tab traversal, that is, by using the Tab key to move to the next control and Shift+Tab to move to the previous control. In most cases, when a control has focus, the action can then be initiated by pressing Enter.

Some complex components use arrow keys to navigate after the component receives focus using the Tab key.

Tab Traversal Sequence on a Page

Default Tab traversal order for a page is from left to right and from top to bottom, as shown in [Figure C-1](#). Tab traversal in a two-column form layout does not follow this pattern, but rather follows a columnar pattern. On reaching the bottom, the tab sequence repeats again from the top.

Figure C-1 Tab Traversal Sequence on a Page



Avoid using custom code to control the tab traversal sequence within a page, as the resulting pages would be too difficult to manage and would create an inconsistent user experience across pages in an application and across applications.

To improve keyboard navigation efficiency for users, you should set the `initialFocusId` attribute on the document. For accessibility purposes, you should also define a `skipLinkTarget` and include a `skip` navigation link at the top of the page, which should navigate directly to the first content-related tab stop.

Tab Traversal Sequence in a Table

The Tab traversals in a table establishes a unique row-wise navigation pattern when the user presses the Tab key to navigate sequentially from one cell to another. One of the following actions can occur when user presses the Enter key in a table cell:

- Clicks on an editable cell and presses Enter key—the focus moves to the editable cell below in the same column in the next row
- Clicks on an editable cell, edits the contents of the cell, and presses Enter key—the focus completes the action in the current cell and moves to the editable cell below in the same column in the next row
- Clicks on an editable cell and presses Tab key without editing the cell, once or more than once, and then presses Enter key—the focus moves to the editable cell below in the next row, in the same column where the user started pressing the tab key.
- Clicks on an editable cell, edits the contents of the cell, and presses Tab key, once or more than once, then presses Enter key—the focus completes the action in the current cell where the user started pressing the Tab key. Focus traverses through the cells sequentially per the number of times the Tab key is pressed. Then, the focus moves to the editable cell below in the next row, in the same column where the user started pressing the tab key
- Clicks on a non-editable cell and presses Enter key—the focus moves to the first editable cell in the next row.
- Clicks on a non-editable cell and presses Tab key, once or more than once, then presses Enter key—the focus traverses through the cells sequentially per the number of times the Tab key is pressed and moves to the first editable cell in the next row.
- Clicks on a cell that contains any command, such as menu, link, or a dialog box, then presses Enter key—the default action for that command is executed

 **Note:**

When user uses the Tab key to traverse through the cells sequentially and presses Enter key to move to the next row, a navigation pattern is formed based on the first set of Tab keys, which is followed in subsequent rows. The navigational pattern is not recognized if arrow keys are used to navigate from one cell to another.

Figure C-2 shows an example of a Tab and Enter keys traversal sequence in a table.

Figure C-2 Tab and Enter Keys Traversal Sequence in a Table

ClickToEdit Table Demo 1

Name	commandLink	inputText	* Required field	inputComboBoxListOf	inputDate
·	Click Me	test	07/12/2004		7/12/2004
..	Click Me	0 B	07/12/2004		7/12/2004
admin.jar	Click Me	1 KB	05/11/2004		5/11/2004
applib	Click Me	0 B	07/12/2004		7/12/2004

ClickToEdit Table Demo 2

Name	commandLink	inputText	* Required field	inputComboBoxListOf	inputDate
·	Click Me	test	07/12/2004		7/12/2004
..	Click Me	0 B	07/12/2004		7/12/2004
admin.jar	Click Me	1 KB	05/11/2004		5/11/2004
applib	Click Me	0 B	07/12/2004		7/12/2004

ClickToEdit Table Demo 3

Name	commandLink	inputText	* Required field	inputComboBoxListOf	inputDate
·	Click Me	test	07/12/2004		7/12/2004
..	Click Me	0 B	07/12/2004		7/12/2004
admin.jar	Click Me	1 KB	05/11/2004		5/11/2004
applib	Click Me	0 B	07/12/2004		7/12/2004

ClickToEdit Table Demo 4

Name	commandLink	inputText	* Required field	inputComboBoxListOf	inputDate
·	Click Me	test	07/12/2004		7/12/2004
..	Click Me	0 B	07/12/2004		7/12/2004
admin.jar	Click Me	1 KB	05/11/2004		5/11/2004
applib	Click Me	0 B	07/12/2004		7/12/2004

In [Figure C-2](#), the user has navigated the rows without editing any cell in the following way:

1. The user clicks a cell in the **inputText** column, giving it focus and making it editable.
Because the Tab key is used to navigate, the **inputText** column is recognized as the starting column for the navigation pattern.
2. The user presses the Tab key and moves the focus in the same row to the cell of the *** Required field** column.
3. The user presses the Tab key and moves the focus in the same row to the cell of the **inputComboListOf** column.
4. The user presses the Enter key and the focus shifts to the **inputText** column in the next row.

Shortcut Keys

While it is possible to use the tab key to move from one control to the next in an ADF Faces application, keyboard shortcuts like the accelerator and access keys are more convenient and efficient. They help users to navigate around the web application easily and access a menu or function quickly.

There are various keyboard shortcuts provided by ADF Faces itself, as well as component attributes that enable you to create specific keyboard shortcuts for your specific applications. ADF Faces categorizes shortcut keys for components into two types, accelerator keys and access keys.

Note:

It is the application developer's responsibility to provide user assistance to identify the application's available keyboard shortcuts. Because an ADF component's terminology can conflict with an application's terminology, the application should also provide a popup or other help that details the keyboard shortcuts available for that component.

Accelerator Keys

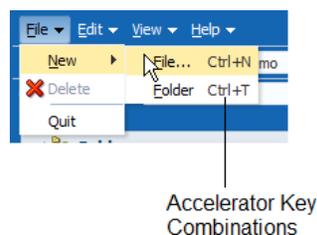
Accelerator keys bypass menu and page navigation and perform actions directly. Accelerator keys are sometimes also called **hot keys**. Common accelerator keys in a Windows application, such as Internet Explorer, are Ctrl+O for Open and Ctrl+P for Print.

Accelerator keys are single key presses (for example, Enter and Esc) or key combinations (for example, Ctrl+A) that initiate actions immediately when activated. A key combination consists of a meta key and an execution key. The meta key may be Ctrl (Command on a Macintosh keyboard), Alt (Option on a Macintosh keyboard), or Shift. The execution key is the key that is pressed in conjunction with the meta key.

Some ADF Faces components have their own built-in accelerator keys. For example, Ctrl+Alt+M is the accelerator key to open the context menu. For more information about ADF Faces components with their own built-in accelerator keys, see the component tag documentation.

ADF Faces also enable you to provide custom accelerator keys to specific menu items, as shown in [Figure C-3](#). All assigned menu accelerator keys are visible when you open the menu.

Figure C-3 Accelerator Keys in a Menu



When defining accelerator keys, you must follow these guidelines:

- Because accelerator keys perform actions directly, if a user presses an accelerator key unintentionally, data may be lost or incorrect data may be entered. To reduce the likelihood of user error, accelerator keys should be used sparingly, and only for frequently and repetitively used functions across applications. As a general rule, less than 25% of available functions should have accelerator keys.
- Custom accelerator keys must not override accelerator keys that are used in the menus of ADF Faces-supported browsers (see the browser and system requirements for supported operating systems and browsers in ADF Faces), and must not override accelerator keys that are used in assistive technologies such as screen readers.
- Custom menu accelerator keys must always be key combinations. The meta key may be Ctrl, Ctrl+Shift, or Ctrl+Alt. Ctrl+Alt is the most used metakey because Ctrl and Ctrl+Shift are commonly used by browsers. The execution key must be a printable character (ASCII code range 33-126).
- Custom menu accelerator keys must be unique. If a page were to have different components that used the same accelerator, it would be difficult for the browser to predict which actions would be executed by the accelerator at any given time.

 **Note:**

In Windows, users have the ability to assign a Ctrl+Alt+*character key* sequence to an application desktop shortcut. In this case, the key assignment overrides browser-level key assignments. However, this feature is rarely used, so it can generally be ignored.

Certain ADF Faces components have built-in accelerator keys that apply when the component has focus. Of these, some are reserved for page-level components, whereas others may be assigned to menus when the component is not used on a page. [Table C-1](#) lists the accelerator keys that are already built into page-level ADF Faces components. You must not use these accelerator keys at all.

Table C-1 Accelerator Keys Reserved for Page-Level Components

Accelerator Key	Used In	Function
Ctrl+Alt+W	Pop-up	Toggle focus between open popups.
Ctrl+Shift+W	Messaging Secondary Windows	
Ctrl+Alt+P	Splitter	Give focus to splitter bar.

The menu commands take precedence if they are on the same page as page-level components, and have the same accelerator keys. For this reason, you must not use the accelerator keys listed in [Table C-3](#) and [Table C-7](#) in menus when the related component also appears on the same page.

Access Keys

Access keys move the focus to a specific UI element, and is defined by the `accessKey` property of the ADF Faces component.

Access keys relocate cursor or selection focus to specific interface components. Every component on the page with definable focus is accessible by tab traversal (using Tab and Shift+Tab); however, access keys provide quick focus to frequently used components. Access keys must be unique within a page.

The result of triggering an access key depends on the associated element and the browser:

- **Buttons:** In both Firefox and Internet Explorer, access keys give focus to the component and directly execute the action. Note that in Internet Explorer 7 access key gives focus to the component, but does not execute the action.
- **Links:** In Firefox, access keys give focus to the component and directly navigate the link; in Internet Explorer, access keys give focus only to the link.
- **Other Elements:** In both browsers, access keys give focus only to the element. For checkbox components, the access key toggles the checkbox selection. For option buttons, the access key performs selection of the option button.

Note that the access key could be different for different browsers on different operating systems. You must refer to your browser's documentation for information about access keys and their behavior. [Table C-2](#) lists access key combinations for button and anchor components in some common browsers.

Table C-2 Access Key For Various Browsers

Browser	Operating System	Key Combination	Action
Google Chrome	Linux	Alt + mnemonic	Click
Google Chrome	Mac OS X	Control + Option + mnemonic	Click
Google Chrome	Windows	Alt + mnemonic	Click
Mozilla Firefox	Linux	Alt + Shift + mnemonic	Click
Mozilla Firefox	Mac OS X	Control + mnemonic	Click
Mozilla Firefox	Windows	Alt + Shift + mnemonic	Click
Microsoft Internet Explorer 7	Windows	Alt + mnemonic	Set focus
Microsoft Internet Explorer 8	Windows	Alt + mnemonic	Click or set focus
Apple Safari	Windows	Alt + mnemonic	Click
Apple Safari	Mac OS X	Control + Option + mnemonic	Click

 **Note:**

- Different versions of a browser might behave differently for the same access key. For example, using Alt + mnemonic for a button component in Internet Explorer 7 sets focus on the component, but it triggers the click action in Internet Explorer 8.
- In Firefox, to change the default behavior of the component when access key combination is used, change the configuration setting for the `accessibility.accesskeycausesactivation` user preference.
- Some ADF Faces components that are named as Button do not use HTML button elements. For example, `af:button` uses an anchor HTML element.

If the mnemonic is present in the text of the component label or prompt (for example, a menu name, button label, or text box prompt), it is visible in the interface as an underlined character, as shown in [Figure C-4](#). If the character is not part of the text of the label or prompt, it is not displayed in the interface.

Figure C-4 Access Key



When defining access keys, you must follow these guidelines:

- Access keys may be provided for buttons and other components with a high frequency of use. You may provide standard cross-application key assignments for common actions, such as Save and Cancel. Each of these buttons is assigned a standard mnemonic letter in each language, such as S for Save or C for Cancel.
- A single letter or symbol can be assigned only to a single instance of an action on a page. If a page had more than one instance of a button with the same mnemonic, users would have no way of knowing which button the access key would invoke.
- Focus change initiated through access keys must have alternative interactions, such as direct manipulation with the mouse (for example, clicking a button).
- The mnemonic must be an alphanumeric character — not a punctuation mark or symbol — and it must always be case-insensitive. Letters are preferred over numbers for mnemonics.
- In Internet Explorer, application access keys override any browser-specific menu access keys (such as Alt+F for the File menu), and this can be a usability issue for users who habitually use browser access keys. Thus, you must not use access keys that conflict with the top-level menu access keys in ADF Faces-supported browsers (for example, Alt+F, E, V, A, T, or H in the English version of Internet Explorer for Windows XP).
- You are responsible for assigning access keys to specific components. When choosing a letter for the access key, there are a few important considerations:

- **Ease of learning:** Although the underlined letter in the label clearly indicates to the user which letter is the access key, you should still pick a letter that is easy for users to remember even without scanning the label. For example, the first letter of the label, like Y in Yes, or a letter that has a strong sound when the label is read aloud, such as x in Next.
- **Consistency:** It is good practice to use the same access key for the same command on multiple pages. However, this may not always be possible if the same command label appears multiple times on a page, or if another, more frequently used command on the page uses the same access key.
- **Translation:** When a label is translated, the same letter that is used for the access key in English might not be present in the translation. Developers should work with their localization department to ensure that alternative access keys are present in component labels after translation. For example, in English, the button **Next** may be assigned the mnemonic letter x, but that letter does not appear when the label is translated to **Suivantes** in French. Depending on the pool of available letters, an alternative letter, such as S or v (or any other unassigned letter in the term Suivantes), should be assigned to the translated term.

 **Note:**

For translation reasons, you should specify access keys as part of the label. For example, to render the label **Cancel** with the C access key, you should use `&Cancel` in the `textAndAccessKey` property (where the ampersand denotes the mnemonic) rather than `C` in the `accessKey` property. Product suites must ensure that access keys are not duplicated within each supported language and do not override access keys within each supported browser unless explicitly intended.

Shortcut Keys for Common Components

[Table C-3](#) lists the shortcut keys assigned to common components such as Menu, Menu bar, Multi-Select Choice List, Multi-Select List Box, and so on.

Table C-3 Shortcut Keys Assigned to Common Components

Shortcut Key	Components	Function
Enter Spacebar	All components	Activate the component, or the component element that has the focus.
Tab Shift+Tab	All components Flash components like ThematicMap, Graph, and Gauge	Move focus to next or previous editable component.
Ctrl+A	All components	Select all.
Alt+Arrow Down	Multi-Select Choice List Multi-Select List Box	Open the list. Use arrow keys to navigate, and press Enter or Spacebar to select.

Table C-3 (Cont.) Shortcut Keys Assigned to Common Components

Shortcut Key	Components	Function
Ctrl+Shift+Home	Multi-Select Choice List	Select all items from top to current selection, or select all items from current selection to bottom.
Ctrl+Shift+End	Multi-Select List Box	
Arrow Left	Menu Bar	Move focus to different menu on a menu bar.
Arrow Right	Splitter	
	Input Number Slider	Move splitter left or right when it is in focus.
	Input Range Slider	
	Input Number Spinbox	Move slider left or right when input number slider or input range slider is in focus.
		Increment or decrement the value when input number spinbox is in focus.
Arrow Up	Menu	Move focus to different menu items in a menu.
Arrow Down	Splitter	
	Input Number Slider	Move splitter up or down when it is in focus.
	Input Range Slider	
		Move slider up or down when input number slider or input range slider is in focus.

Shortcut Keys for Widgets

[Table C-4](#) lists the shortcut keys assigned to common widgets such as Disclosure control, Hierarchy control, and Dropdown lists.

Table C-4 Shortcut Keys Assigned to Common Widgets

Shortcut Key	Components	Function
Enter	Disclosure Control	Open a closed Disclosure control, or close a open Disclosure control. A disclosure control is an icon that indicates that more content is available to either be shown or hidden.
Arrow Down/Arrow Up		
Ctrl+Alt+R	Active Data	Applicable only if the page contains active data.
Ctrl+Shift+^	Hierarchy Control	If in hierarchy viewer, open the hierarchy popup.
Alt+Down Arrow	Dropdown list	Open the dropdown list.
Enter	Dropdown list	Select the focussed option of dropdown list.
Ctrl+A	Multi-Select List Box	Select all options.
Ctrl+Shift+Home	Multi-Select List Box	Select all options from the first option to the current option.

Table C-4 (Cont.) Shortcut Keys Assigned to Common Widgets

Shortcut Key	Components	Function
Ctrl+Shift+End	Multi-Select List Box	Select all options from the current option to the last option.
Ctrl+Alt+M	Various components	Opens the context menu in components that support it, such as Calendar and Table.
Ctrl+Shift+W Ctrl+Alt+W	Various components	Toggle between open detachable menus.
Ctrl+Alt+P	Splitter	Move focus to next Splitter component.
Enter	Splitter	If the Splitter is in focus, toggles the split section from closed to open state.
Ctrl+Alt+F4	Tab	Remove the tab, if it is removable.

Shortcut Keys for Rich Text Editor Component

[Table C-5](#) lists shortcut keys assigned to the Rich Text Editor component. In regular mode, all toolbar controls appear on top of the Rich Text Editor area.

Table C-5 Shortcut Keys Assigned to Rich Text Editor Component

Shortcut Key	Components	Function
Ctrl+B	Rich Text Editor	Boldface
Ctrl+I	Rich Text Editor	Italics
Ctrl+U	Rich Text Editor	Underline
Ctrl+5	Rich Text Editor	Strikethrough
Ctrl+E	Rich Text Editor	Center alignment
Ctrl+J	Rich Text Editor	Full-justified alignment
Ctrl+L	Rich Text Editor	Left alignment
Ctrl+R	Rich Text Editor	Right alignment
Ctrl+H	Rich Text Editor	Create hyperlink
Ctrl+M	Rich Text Editor	Increase indentation
Ctrl+Shift+M	Rich Text Editor	Decrease indentation
Ctrl+Shift+H	Rich Text Editor	Remove hyperlink
Ctrl+Shift+L	Rich Text Editor	Bulleted list
Ctrl+Alt+L	Rich Text Editor	Numbered list
Ctrl+Shift+S	Rich Text Editor	Clear text styles
Ctrl+Alt+-	Rich Text Editor	Subscript
Ctrl+Alt++	Rich Text Editor	Superscript
Ctrl+Alt+R	Rich Text Editor	Enable rich text editing mode
Ctrl+Alt+C	Rich Text Editor	Enable source code editing mode

Table C-5 (Cont.) Shortcut Keys Assigned to Rich Text Editor Component

Shortcut Key	Components	Function
Ctrl+Y	Rich Text Editor	Redo
Ctrl+Z	Rich Text Editor	Undo

Shortcut Keys for Table, Tree, and Tree Table Components

[Table C-6](#) lists shortcut keys assigned to Table, Tree, and Tree Table. For information about Tables and Trees, see [Using Tables, Trees, and Other Collection-Based Components](#).

Table C-6 Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Tab	Table	Move focus to next or previous cell or editable component.
Shift+Tab	Tree Table	In a table, navigate to the next or previous editable content in cells in left-to-right direction. If the focus is on the last cell of a row in the table, the Tab key moves focus to the first editable cell in the next row. Similarly, Shift + Tab moves focus to the previous row.
Ctrl+A	Table Tree Table	Select all components, including column headers, row headers, and data area.
Ctrl+Alt+M	Table Tree Tree Table	Launch context menu. You can also launch context menu by pressing Ctrl+Alt+B.
Ctrl+Shift+^	Tree Tree Table	Go up one level.
Ctrl+Arrow Right	Table Tree Tree Table	In a table, expand row. In a tree or tree table, expand nodes or detailStamp facets.
Ctrl+Arrow Left	Table Tree Tree Table	In a table, collapse row. In a tree or tree table, collapse nodes or detailStamp facets.

Table C-6 (Cont.) Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Enter	Table	Navigate to the next editable cell or previous editable cell of the column.
Shift+Enter	Tree Tree Table	In a table, navigate to the next or previous editable content in cells in top-to-bottom direction. If focus is on the column header, sort table data in ascending order. Pressing Enter again sorts the column in descending order. If the focus is on the filter cell, perform table filtering. In a table, if the user presses Tab key to navigate from one cell to another and presses Enter, move focus to the next row to follow same navigational pattern. See Tab Traversal Sequence in a Table .
Arrow Left	Table	Move focus.
Arrow Right	Tree Table	In a table, when the focus is on an editable component, move the text cursor.

Table C-6 (Cont.) Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Arrow Up	Table	Move focus.
Arrow Down	Tree Table	<p>If a row is selected, move focus to the previous row or next row. If no row is selected, scroll the table one row up or down.</p> <p>In a table, when the focus is on an editable component that supports multiple options (such as <code>selectOneChoice</code> and <code>inputNumberSpinBox</code>), scroll the selected option.</p> <p>If the first row is selected, move focus to the column header.</p> <p>In an editable table, if the user clicks a cell with an editable component (such as a text box, or a checkbox), a button or a link component, focus is set to the component in the cell. To use Up and Down arrow keys for navigation, focus should be moved from the editable component to the cell. The user would need to click on the background of the same cell (or any cell of the same row) again to move the focus.</p> <p>Note: If <code>selectionEventDelay</code> is enabled, row selection during keyboard navigation is delayed by 300ms to allow table keyboard navigation without causing unwanted row selection.</p>
Ctrl+Arrow Up	Table	Move focus.
Ctrl+Arrow Down		<p>If in edit mode, submit the changes made in the current row and navigate to the previous row or next row.</p> <p>In the click-to-edit table, when the focus is on an editable component that supports multiple options (such as <code>selectOneChoice</code> and <code>inputNumberSpinBox</code>), scroll the selected option.</p>
Ctrl+Arrow Left	Table	Move focus.
Ctrl+Arrow Right		If in edit mode, when the focus is on an editable component, move the text cursor.
Shift+Arrow Left	Table	Move focus and add to selection.
Shift+Arrow Right	Tree Table	
Ctrl+Shift+Arrow Left	Table	Move the selected column to the left or right.
Ctrl+Shift+Arrow Right	Tree Table	

Table C-6 (Cont.) Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Shift+Arrow Up	Table	Select multiple rows.
Shift+Arrow Down	Tree Table Tree	
Page Up	Table	If a row is selected, scroll and select the same row of the next or previous page.
Page Down	Tree Table	If no row is selected, scroll by one page.
Alt+Page Up	Table	Horizontally scroll the table to the right or left.
Alt+Page Down	Tree Table	
Space Bar	Table	Select the node.
Ctrl+Space Bar	Tree Tree Table	To select or remove multiple nodes, press Ctrl+Space Bar.
Shift+Space Bar	Table Tree Table	Select multiple rows.
Esc	Table Tree Table	Remove selection. If the focus is on the cell, exit click-to-edit mode, revert the cell value to original value, and return focus to the cell. Press Esc key again to move focus to the row header.
F2	Table Tree Table	Activate click-to-edit mode for the row. Press F2 again to disable cell navigation mode.

Shortcut Keys for ADF Data Visualization Components

[Table C-7](#) lists shortcut keys assigned to ADF Data Visualization Components including charts, diagram, Gantt chart, hierarchy viewer components, geographic and thematic maps, NBox, pivot table, and pivot filter bar. For information about ADF Data Visualization Components, see [Introduction to ADF Data Visualization Components](#).

Table C-7 Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Arrow Left	Charts: Area, Bar, Bubble, Combination, Funnel, Line, Pie, Scatter, Spark	Move focus.
Arrow Right	Chart legend with horizontal orientation	If the focus is on the bars in bar charts, move focus and selection to bar on left or bar on right.
	List region of all Gantt chart types	If the focus is on a pie slice in a pie chart, move focus and selection to previous series in a counterclockwise direction or next series in a clockwise direction.
	Project Gantt chart region	
	Scheduling Gantt chart region	If the focus is on a dot, bubble, or bar in an area, bubble, combination, funnel, line, scatter, or spark chart, move focus and selection to the nearest bar, dot, or bubble on left or right.
	Resource Utilization Gantt chart region	
	Geographic and Thematic Map	
	Hierarchy Viewer - nodes	If the focus is on a series in a chart legend, move focus to series on left or series on right.
	Pivot table	
	Pivot filter bar	
	NBox	If the focus is on the chart region of scheduling Gantt, the arrow key navigation selects the previous or next taskbar of the current row.
	Diagram	If the focus is on the time bucket of resource utilization Gantt, the arrow key navigation selects the previous or next time bucket in the current row.
		If the focus is on the ADF geographic map, the arrow key navigation pans left or right by a small increment. Press Home or End key to pan by a large increment.
		If the focus is on the node component of ADF hierarchy viewer, press Ctrl+Arrow to move the focus left or right without selecting the component.
		If you are using arrow keys to navigate cells of an editable pivot table, each focused cell is activated for editing before allowing you to navigate to the next cell, making the navigation slower. Press the Esc key to deactivate the edit mode of the focused cell, and navigate faster. To edit a cell, press the F2 or Enter key.
		If the focus is on the pivot table data cell, press Ctrl+Arrow Left to jump to the corresponding row header cell. If the locale is bidirectional (such as Arabic), press Ctrl+Arrow Right to jump to the corresponding row header cell.

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
		If the focus is on an NBox cell or node, move focus and selection to the next or previous cell, parent node (sorted by size in descending order) or individual node. Node navigation is based on list navigation; down or right moves to the next element and up or left moves to the previous element.

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Arrow Up	Charts: Area, Bar (Stacked), Bubble, Combination, Funnel, Horizontal Bar, Line, Scatter, Spark	Move focus.
Arrow Down	Chart legend with vertical orientation	If the focus is on the bars in horizontal bar charts, move focus and selection up or down to next or previous bar.
	List region of all Gantt chart types	If the focus is on a stacked bar chart, move focus and selection up or down to next or previous series on the same bar.
	Project Gantt chart region	If the focus is on a dot, bubble, or bar in an area, bubble, combination, funnel, line, scatter, or spark chart, move focus and selection up or down to the nearest bar, dot, or bubble.
	Scheduling Gantt chart region	If the focus is on a series in a chart legend, move focus up or down to next or previous series.
	Resource Utilization Gantt chart region	If the focus is on the chart region of project Gantt, the arrow key navigation selects previous or next row.
	Geographic and Thematic Map	If the focus is on the chart region taskbar of scheduling Gantt, the arrow key navigation selects the first taskbar of the previous row or the next row.
	Hierarchy Viewer - nodes	If the focus is on the time bucket of resource utilization Gantt, the arrow key navigation selects the time bucket of the previous row or next row.
	Pivot table	If the focus is on the ADF geographic map component, the arrow key navigation pans up or down by a small increment.
	Pivot filter bar	If the focus is on the node component of ADF hierarchy viewer, press Ctrl+Arrow keys to move the focus up or down without selecting the component.
	NBox	If you are using arrow keys to navigate cells of an editable pivot table, each focused cell is activated for editing before allowing you to navigate to the next cell, making the navigation slower. Press the Esc key to deactivate the edit mode of the focused cell, and navigate faster. To edit a cell, press the F2 or Enter key.
		If the focus is on the pivot table data cell, press Ctrl+Arrow Up to jump to

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
		the corresponding column header cell.
Page Up Page Down	Chart legend with vertical orientation Chart plot area Geographic and Thematic Map Hierarchy Viewer - diagram	If the focus is on an NBox cell or node, move focus and selection up or down to the nearest cell, parent node (sorted by size in descending order) or individual node. Node navigation is based on list navigation; down or right moves to the next element and up or left moves to the previous element. If the focus is on a chart legend, scroll up or down. If the focus is on a chart plot area, pan up or down. If the focus is on the geographic map component, the page key navigation pans up or down by a large increment. If the focus is on the diagram of a hierarchy viewer, press and hold to Page Up or Page Down keys to pan up or down. Press Shift+Page Up or Shift+Page Down to pan left or right. Press and hold Shift+Page Down to pan continuously.
+	Geographic and Thematic Map Hierarchy Viewer - diagram	Increase zoom level. If the focus is on the diagram of a hierarchy viewer, press number keys 1 through 5 to zoom from 10% through 100%. Press 0 to zoom the diagram to fit within available space. Press and hold to continuously increase zoom.
-	Geographic and Thematic Map Hierarchy Viewer - diagram	Decrease zoom level. If the focus is on the diagram of a hierarchy viewer, press number keys 1 through 5 to zoom from 10% through 100%. Press 0 to zoom the diagram to fit within available space. Press and hold to continuously decrease zoom.
Ctrl+Alt+M	All Gantt chart types Pivot table Pivot filer bar	Launch context menu.

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Ctrl+Left Arrow Ctrl+Right Arrow	Charts: Area, Bar, Bar (Stacked), Bubble, Funnel, Horizontal Bar, Line, Pie, Scatter, Spark NBox	Move focus to nearest bar, dot, or bubble to the left or right of the current selection, but do not select. If the focus is on a pie slice in a pie chart, move focus to previous series in a counterclockwise direction or next series in a clockwise direction, but do not select. If the focus is on a series in a stacked bar chart, move focus to nearest series to the left or right of the selected series, but do not select. If the focus is on an NBox node, move focus without selection.
Ctrl+Up Arrow Ctrl+Down Arrow	Charts: Area, Bar, Bar (Stacked), Bubble, Combination, Funnel, Horizontal Bar, Line, Scatter, Spark NBox	Move focus and to nearest bar, dot, or bubble above or below the current selection, but do not select. If the focus is on a series in a stacked bar chart, move focus to nearest series above or below the selected series, but do not select. If the focus is on an NBox node, move focus without selection.
Ctrl+Spacebar	Charts: Area, Bar, Bar (Stacked), Bubble, Combination, Funnel, Horizontal Bar, Line, Pie, Scatter, Spark NBox	Move focus and to nearest bar, dot, or bubble above or below the current selection, but do not select. If the focus is on a series in a stacked bar chart, move focus to nearest series above or below the selected series, but do not select. If the focus is on an NBox node, select or multi-select.
Shift+Left Arrow Shift+Right Arrow	Charts: Area, Bar, Bubble, Combination, Funnel, Horizontal Bar, Line, Pie, Scatter, Spark NBox	Move focus and multi-select nearest bar, dot, or bubble to the left or right of the current selection. If the focus is on a pie slice in a pie chart, move focus and multi-select previous series in a counterclockwise direction or next series in a clockwise direction. If the focus is on a series in a stacked bar chart, move focus and multi-select the nearest series to the left or right of the selected series. Move focus and multi-select nearest NBox node left or right.

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Shift+Up Arrow Shift+Down Arrow	Charts: Area, Bar (Stacked), Bubble, Combination, Funnel, Horizontal Bar, Line, Scatter, Spark NBox	Move focus and multi-select nearest bar, dot, or bubble above or below the current selection. Move focus and multi-select the nearest NBox node up or down.
Home	Hierarchy Viewer - nodes	Move focus to first node in the current level.
End	Hierarchy Viewer - nodes	Move focus to last node in the current level.
Ctrl + Home	Hierarchy Viewer - nodes	Move focus and select the root node.
<	Hierarchy Viewer - nodes	Switches to the active node's previous panel.
>	Hierarchy Viewer - nodes	Switches to the active node's next panel.
Ctrl + Enter	Hierarchy Viewer - nodes	Toggle the display of the children of the active node.
Ctrl + /	Hierarchy Viewer - nodes	Synchronize all nodes to display the active node's panel.
Ctrl+Shift+^	Hierarchy Viewer - nodes	Go up one level.
Ctrl+/	Hierarchy Viewer - nodes	Switch content panel.
Ctrl+Alt+0	Hierarchy Viewer - diagrams	Center the active node and zoom the diagram to 100%.
Tab	Hierarchy Viewer - nodes Pivot table Pivot filter bar NBox	Move focus through elements. From a component outside an NBox, move focus from NBox, to legend, and then to next component. Use Shift+Tab to move focus to legend, to NBox, and then to previous component.
Esc	Hierarchy Viewer - nodes NBox	Return focus to the containing node. If the focus is on search panel, close the panel. Close the Detail window, if it appears while hovering over a node. Drill up NBox cell or category node.
Spacebar	Hierarchy Viewer - nodes Pivot table Pivot filter bar	Select the active node. Press Ctrl +Spacebar to toggle selection of the active node, and for selecting multiple nodes.
Enter	Hierarchy Viewer - nodes Pivot table Pivot filter bar NBox	Isolate and select active node. Press Shift+Enter to toggle the state of the node. Drill down NBox category node.
/	Hierarchy Viewer - nodes	Toggle control panel state.

Table C-7 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
[NBox	Move focus and selection to the first node in the cell or container.
]	NBox	Move focus and selection from the node to the parent container.
Ctrl+F	Hierarchy Viewer - nodes	If the ADF hierarchy viewer component is configured to support search functionality, open the search panel.
Ctrl+Alt+1 through Ctrl+Alt+5	Hierarchy Viewer - nodes	Switch diagram layout.
Shift+Alt+Arrow keys	Pivot table Pivot filter bar	Change the layout by pivoting a row, column, or filter layer to a new location. Use Shift+Alt+Arrow keys to perform the following: <ul style="list-style-type: none"> • Provide visual feedback, showing potential destination of the pivot operation, if the header layer is selected • Select different destination locations. • Moving or swapping the selected header layer to the specified destination.

Some ADF Data Visualization Components provide some common functions to the end user through menu bar, toolbar, context menu, or a built-in Task Properties dialog box. You may choose to show, hide, or replace these functionality. If you hide or replace any functionality, you must provide alternate keyboard accessibility to those functions.

Shortcut Keys for Calendar Component

The Calendar component has several views: Day view, Week view, Moth view, and List view.

[Table C-8](#) lists shortcut keys assigned to the Calendar component.

Table C-8 Shortcut Keys Assigned to Calendar Component

Shortcut Key	Components	Function
Tab Shift+Tab	Calendar	<p>Move focus.</p> <p>If the focus is on the calendar toolbar, move focus through Day, Week, Month, List, Forward button, Backward button, and Today button.</p> <p>In the day view, move focus through activities of the day.</p> <p>In the week view and month view, move focus through the Month Day header labels only. Use Arrow keys to navigate through activities, "+n more links", and Month Day header labels.</p> <p>In the month view, if the focus is on a Month Day header label at the end of the week, move focus to the Month Day header label of the following week.</p> <p>In the list view, move focus to the day, and then through the activities of the day.</p>
Arrow Left Arrow Right	Calendar	<p>Move focus.</p> <p>In the day view, Right and Left arrows do not move focus.</p> <p>In the week view, if the focus is on an activity, move focus to the first activity of the previous or next day. If the previous or next days contain no activities, move focus to the day header.</p> <p>In the month view, the following interaction occurs:</p> <ul style="list-style-type: none"> • If the focus is on a Month Day header label, move focus to the previous or next day label. <p>If the focus is on the label of the last day of the week in the first week of the month, Right Arrow moves focus to the label of the first day of the week in the second week of the month. If the focus is on the label of the last day of the month, the Right Arrow does nothing.</p> • If the focus is on an activity, move focus to the next activity of the previous or next day. If the previous or next day does not contain any activities, move focus to the Month Day label. If focus is on an activity in the last day of a week, the Right Arrow does nothing. • If the focus is on a "+n more" link, move focus to the next "+n more" links, if they exist. <p>If adjacent "+n more" links do not exist, move focus to the last activity of the day. If the "+n more" link resides in a day at the beginning or end of the week, the Left or Right Arrow do nothing.</p>

Table C-8 (Cont.) Shortcut Keys Assigned to Calendar Component

Shortcut Key	Components	Function
Arrow Up Arrow Down	Calendar	<p>Move focus.</p> <p>In the day view, move focus through activities. When activities conflict and appear within the same time slot, the Down Arrow moves focus right and the Up Arrow moves focus left.</p> <p>In the week view, move focus through activities of the day. If the focus is on the first activity of a day, the Up Arrow moves focus to the day header. If the focus is on the day header, the Down Arrow moves focus to the first activity of that day. If the day has no activities, the Down Arrow does nothing.</p> <p>In the month view, move focus through activities in a day.</p> <ul style="list-style-type: none"> • If the focus is on the first activity in a day, the Up Arrow moves focus to the Month day header label. • If the focus is on the Month Day header label, the Up Arrow moves focus to the last activity of the day above it. • If the focus is on the last activity on a day in the last week of the month, the Down Arrow does nothing. • If the focus is on the month header day label in the first week of the month, the Up Arrow does nothing.
Ctrl+Alt+M	Calendar	<p>Launch context menu.</p> <p>You can also launch context menu by pressing Ctrl+Alt+B.</p>

 **Note:**

When using arrows to navigate through activities of a month or week, all-day activities get focus only when the user is navigating within a day, which an all-day activity starts on. Otherwise, all-day activities are skipped.

Default Cursor or Focus Placement

When a user opens an input form (built with ADF Faces) to enter some data, the default cursor is placed on the most suitable component so that the user can proceed with the help of a keyboard instead of using a mouse first. After the initial focus is set, the user can take control on cursor position.

The default cursor puts the initial focus on a component so that keyboard users can start interacting with the page without excessive navigation.

Focus refers to a type of selection outline that moves through the page when users press the tab key or access keys. When the focus moves to a field where data can be

entered, a cursor appears in the field. If the field already contains data, the data is highlighted. In addition, after using certain controls (such as a list of values (LOV) or date-time picker), the cursor or focus placement moves to specific locations predefined by the component.

During the loading of a standard ADF Faces page, focus appears on the first focusable component on the page — either an editable widget or a navigation component. If there is no focusable element on the page, focus appears on the browser address field.

When defining default cursor and focus placement, you should follow these guidelines:

- ADF Faces applications should provide default cursor or focus placement on most pages so that keyboard users have direct access to content areas, rather than having to tab through UI elements at the top of the page.
- You can set focus on a different component than the default when the page is loaded. If your page has a common starting point for data entry, you may change default focus or cursor location so that users can start entering data without excessive keyboard or mouse navigation. Otherwise, do not do this because it makes it more difficult for keyboard users (particularly screen reader users) to orient themselves after the page is loaded.

The Enter Key

The Enter key in an ADF Faces application causes a command line, form, or dialog box to operate its default function. It is typically used to finish an input form and begin the desired process.

The Enter key triggers an action when the cursor is in certain fields or when focus is on a link or button. You should use the Enter key to activate a common commit button, such as in a Login form or in a dialog.

Many components have built-in actions for the Enter key. Some examples include:

- When focus is on a link or button, the Enter key navigates the link or triggers the action.
- When the cursor is in a query search region, quick query search, or Query-By-Example (QBE) field, the Enter key triggers the search.
- When in a table, pressing the Enter key triggers one of the following actions:
 - Clicks on an editable cell and presses Enter key—the focus moves to the editable cell below in the same column in the next row
 - Clicks on an editable cell, edits the contents of the cell, and presses Enter key—the focus completes the action in the current cell and moves to the editable cell below in the same column in the next row
 - Clicks on an editable cell and presses Tab key without editing the cell, once or more than once, and then presses Enter key—the focus moves to the editable cell below in the next row, in the same column where the user started pressing the tab key.
 - Clicks on an editable cell, edits the contents of the cell, and presses Tab key, once or more than once, then presses Enter key—the focus completes the action in the current cell where the user started pressing the Tab key. Focus traverses through the cells sequentially per the number of times the Tab key is

pressed. Then, the focus moves to the editable cell below in the next row, in the same column where the user started pressing the tab key

- Clicks on a non-editable cell and presses Enter key—the focus moves to the first editable cell in the next row.
- Clicks on a non-editable cell and presses Tab key, once or more than once, then presses Enter key—the focus traverses through the cells sequentially per the number of times the Tab key is pressed and moves to the first editable cell in the next row.
- Clicks on a cell that contains any command, such as menu, link, or a dialog box, then presses Enter key—the default action for that command is executed

 **Note:**

When user uses the Tab key to traverse through the cells sequentially and presses Enter key to move to the next row, a navigation pattern is formed based on the first set of Tab keys, which is followed in subsequent rows. The navigational pattern is not recognized if arrow keys are used to navigate from one cell to another.

D

Creating Web Applications for Touch Devices Using ADF Faces

This appendix describes how to implement web-based applications for touch devices. This appendix includes the following sections:

- [About Creating Web Applications for Touch Devices Using ADF Faces](#)
- [How ADF Faces Behaves in Mobile Browsers on Touch Devices](#)
- [Best Practices When Using ADF Faces Components in a Mobile Browser](#)

About Creating Web Applications for Touch Devices Using ADF Faces

Oracle JDeveloper enables developers to rapidly develop applications that run on multiple touch devices. With the help of the ADF Faces framework, application developers can quickly develop applications for multiple mobile platforms such as iOS, Android, and Windows.

The ADF Faces framework is optimized to run in mobile browsers such as Safari, Chrome, and Microsoft Edge. The framework recognizes when a mobile browser on a touch device is requesting a page, and then delivers only the JavaScript and peer code applicable to a mobile device. However, while a standard ADF Faces web application will run in mobile browsers, because the user interaction is different and because screen size is limited, when your application needs to run in a mobile browser, you should create touch device-specific versions of the pages.

This appendix provides information on how ADF Faces works in mobile browsers on touch devices, along with best practices for implementing web pages specifically for touch devices.

How ADF Faces Behaves in Mobile Browsers on Touch Devices

Oracle ADF provides specific features to enable web applications developed with ADF Faces to run on touch-based devices. These capabilities include support for touch gesture interactions, adaptive rendering to match mobile browsers capabilities, and certification on common mobile browsers.

In touch devices, users touch the screen instead of clicking the mouse. ADF faces now supports touch events too like touchstart, touchend, touchmove, touchcancel. Advanced drag and drop, tap and hold gestures are also supported. Very basic touch gestures like tap are mapped to mouse events but advanced support is provided for iOS and Android platforms. ADF Faces provides peers to handle the conversion in ADF Faces components. To better handle the conversion differences between touch devices and desktop devices, for each component that needs one, ADF Faces

provides both a touch device-specific peer and a desktop-specific peer (for information about peers, see [About Using ADF Faces Architecture](#)).

These peers allow the component to handle events specific to the device. For example, the desktop peer handles the mouse over and mouse out events, while the touch device peer handles the touch start and touch end events. The base peer handles all common interactions. This separation provides optimized processing on both devices.

The touch device peers provide the logic to simulate the interaction on a desktop using touch-specific gestures. [Table D-1](#) shows how desktop gestures are mapped to touch device gestures.

Table D-1 Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
Click	Tap	Mouse down	Execute a button
Select	Tap	Selected	Select a table row
Multi select	Tap selects one, tap selects another, tapping a selected object deselects it	Selected	Select multiple graph bars
Drag and drop in a simple interface	Finger down + drag	Mouse down	Drag a slider thumb or a splitter
Drag and drop for use cases requiring both drag and drop as well as data tips	Finger down + short hold + drag	Mouse down	Move a task bar in a Gantt chart
Hover to show data tip	Finger down + hold	Hover (mouseover)	Show graph data tip
Hover to show popup	Finger down + hold	Hover (mouseover)	Show a popup from a calendar
Click to dismiss a popup	Tap outside of the popup	Mouse click	Dismiss a popup by clicking outside of it
Line data cursor on graph	Finger down + hold	Hover	Trace along the x-axis of a graph and at the intersection of the y-axis, the data value is displayed in a tip.

Table D-1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
Right-click to launch a context menu	Finger down + hold or finger down + hold + finger up (when gesture conflict exists with another finger down + hold gesture). Finger down + hold + finger up over an area for Thematic Map.		Show graph or calendar activity context menu Context menu on finger up examples: Graph: finger down + hold = data tip; finger up = context menu. Graph (bubble): finger down + hold + move = drag and drop; finger up = context menu. Gantt (task bar): finger down + hold = data tip; finger down + hold + move = drag and drop; finger up = context menu
Pan	One finger swipe (when no conflict with other gestures). Otherwise, two finger swipe	Enabled	Pan map
Zoom in/out	Double tap (browser zoom). When in maximized state, pinch in/out can perform zoom. For Thematic Map, only pinch in/out.	Enabled	Zoom browser screen Zoom graph or map
Double-click Thematic Map component	Double tap	Drilled	Drills into an area in an area data layer.
Hover to show data tip or popup in Thematic Map	Finger down + hold	Hover (mouse-over)	Show area or marker data tip or launch popup.
Double-click to set anchor in the Hierarchy Viewer component	Double tap	When the <code>setAnchorListener</code> has a value, causes the node to be the root of the tree. When the value is not set, double tap causes a browser zoom.	Double tap a node within a hierarchy causes it to become the root node.
Click the isolate icon on the Hierarchy Viewer component	Tap node, then tap isolate icon	Panel card is isolated	Tap the top of the card and then the isolate icon to view only that card and any direct reports.

Table D-1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
Click the collapse icon on the Hierarchy Viewer component	Swipe up on card	Collapsed panel card	Collapse a panel card
Click the expand icon on the Hierarchy Viewer component	Swipe down on card	Expanded panel card	Expand a collapsed panel card.
Hover to show fly out buttons on Hierarchy Viewer	Tap card	Fly out buttons display	Tap a card to display the fly out buttons
Click right or left arrow buttons on Hierarchy Viewer component	Swipe left or right on card	Switch panel cards	Swipe left to view address, or swipe right to view content.
Click navigation buttons to laterally traverse the hierarchy	Swipe left or right on the lateral navigation line, or tap the arrow, or touch and short hold + finger up to display the navigation buttons	Traverse the hierarchy	View more descendants of the root node.
na	Single tap the Maximize icon	Maximizes the component	
na	Double-tap the Maximize icon or double-tab the hierarchy viewer background	Maximizes the component and zooms to fit	
Use circle, square, or polygon tool on a map to drag and select a region	Finger down, draw shape	Selected	Use finger to select an area on a map
Use measurement tool on a map to click start point and end point	Tap measurement tool, finger down, draw line	Line drawn and calculated distance displayed	Use finger to select measurement tool, then tap to select point A and draw line to point B.

Table D-1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
Use area tool on a map to click start point and end point	Tap area tool, finger down, draw line	Line drawn and calculated area displayed	Use finger to select area tool, then tap to select point A and draw line to point B, and so on.

For further optimization, ADF Faces partitions JavaScript, so that the touch device JavaScript is separated from the desktop JavaScript. Only the needed JavaScript is downloaded when a page is rendered. Also, when a touch device is detected, CSS content specific to touch devices is sent to the page. For example, on a touch device, checkboxes are displayed for items in the shuttle components, so that the user can easily select them. On a desktop device, the checkboxes are not displayed.

Using device-specific peers, JavaScript, and CSS allows components to function differently on desktop and touch devices. [Table D-2](#) shows those differences.

Table D-2 Component Differences in Mobile Browsers

Component	Functionality	Difference from desktop component
<code>selectManyShuttle</code> and <code>selectOrderShuttle</code>	Selection	Select boxes are displayed that allow users to select the item(s) to shuttle.
<code>table</code>	Selection	Users select a row by tapping it and unselect a row by tapping it again. Multi-select is achieved simply by tapping the rows to be selected. That is, selecting a second row does not automatically deselect the first row.
<code>table</code>	Scroll	Instead of scroll bars, the table component displays a footer that allows the user to jump to specific pages of rows. The number of rows on a page is determined by the <code>fetchSize</code> attribute.
ADF Faces dialog framework	Windows	When a command component used to launch the dialog framework has its <code>windowEmbedStyle</code> attribute set to <code>window</code> (to launch in a separate window), ADF Faces overrides this value and sets it to <code>inlineDocument</code> , so that the dialog is instead launched inline within the parent window.
<code>menu</code>	Detachable menus	Detachable menus are not supported. The <code>detachable</code> attribute is ignored.

Table D-2 (Cont.) Component Differences in Mobile Browsers

Component	Functionality	Difference from desktop component
inlineFrame	Geometry management	On touch devices, iFrame components ignore dimensions, and are always only as tall as their contents. Therefore, if the inlineFrame is stretched by its parent, the content may be truncated, because scroll bars are not used on touch devices. When the inlineFrame is stretched by its parent, 40 pixels of padding and overflow are added to the inline style.
panelBox and showDetailHeader	Maximize	On touch devices, when showMaximize is set to auto, a maximize icon displays, allowing the user to maximize the component to the full browser window. This icon will not display on desktops.
Various components	Icons, buttons, and links	Icons and buttons are larger and spaces between links are larger to accommodate fingers

Because some touch devices do not support Flash, ADF Faces components use HTML5 for animation transitions and the like. This standard ensures that the components will display on all devices.

Best Practices When Using ADF Faces Components in a Mobile Browser

To build the ADF Faces application for the touch device in an adaptive, interactive and gesture friendly way, you must follow some best practices. This will enable a better user experience and functionalities won't be limited.

When you know that your application will be run on touch devices, the best practice is to create pages specific for that device. You can then use code similar to that of the following example to determine what device the application is being run on, and then deliver the correct page for the device.

```
public boolean isMobilePlatform()
{
    RequestContext context = RequestContext.getCurrentInstance();
    Agent agent = context.getAgent();

    return
    Agent.TYPE_PDA.equals(agent.getType()) ||
    Agent.TYPE_PHONE.equals(agent.getType()) ||
    (
    Agent.AGENT_WEBKIT.equals(agent.getAgentName()) &&
    (
    // iPad/iPhone/iPod touch will come in as a desktop type and an iphone platform:
    "iphone".equalsIgnoreCase(agent.getPlatformName())
    )
    );
}
```

While your touch device application can use most ADF Faces components, certain functionality may be limited, or may be frustrating, on touch devices. [Table D-3](#) provides best practices to follow when developing an application for touch devices.

Table D-3 Best Practices for ADF Faces Components in a Mobile Browser

Component/Functionality	Best Practice
Geometry management	<p>Set the <code>oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS</code> <code>web.xml</code> parameter to <code>auto</code>.</p> <p>This setting ensures that the page will flow instead of stretch. See Geometry Management for Layout and Table Components.</p>
Partial page navigation	<p>Using partial page navigation means that the JavaScript and other client code will not need to be downloaded from page to page, improving performance. See Using Partial Page Navigation.</p>
Navigation	<p>Provide more direct access to individual pieces of content. A good rule is to have only one task per page, instead of using many regions on a page, separated by splitters. For example, instead of using a <code>panelSplitter</code> with a tree in the left pane to provide navigation, provide a list-based navigation model.</p>
ADF task flows	<p>When using ADF Task flows, use page fragments instead of full pages. Page fragments are inserted into a page using regions, which live in the parent document and don't require an <code>iFrame</code>. If you must use full pages, then do not use the dialog framework, and instead run them inline with the page.</p> <p>For information about task flows, see "Getting Started with ADF Task Flows" in <i>Developing Fusion Web Applications with Oracle Application Development Framework</i>.</p>
document tag	<p>You can configure the <code>document</code> tag to reference a large icon (typically 129 pixels by 129 pixels). See How to Configure the document Tag.</p>
Handling touch events on the <code>richTextEditor</code> component	<p>When a user clicks the editable area in the <code>richTextEditor</code> component, the following happens:</p> <ul style="list-style-type: none"> • Editable area has focus • Keyboard is activated • On all versions of iOS, any touch event listeners added to the document are disabled <p>When a user clicks outside the editable area, the following happens:</p> <ul style="list-style-type: none"> • Editable area loses focus (blur) • Keyboard is closed • The cursor position is lost (and will not be retained upon clicking back into the editable area) • On iOS, touch event listeners are added back to the document (on blur) <p>If you add any touch event listeners to the document, you must use the <code>AdfAgent.addBubbleEventListener()</code> to ensure that the touch event listeners are removed and restored properly.</p>
<code>richTextEditor</code> editor window size	<p>Set the <code>rows</code> attribute. If this attribute is not specified, when in WYSIWYG mode, the edit window will grow with the content added.</p>

Table D-3 (Cont.) Best Practices for ADF Faces Components in a Mobile Browser

Component/Functionality	Best Practice
Tables	<p>By default, when rendered on tablet devices, tables display a footer that allows the user to jump to specific pages of rows. However, when you want tables to scroll smoothly using high-water mark scrolling to mimic scrolling on tablets, you should change the default setting for the <code>scrollPolicy</code> attribute. High-water mark scrolling closely resembles the way virtualized touch scrolling (scrolling in both horizontal and vertical directions) behaves on tablets. You may use this setting to address performance problems with virtualized scrolling of table data on tablets.</p> <p>For all tables to display on a tablet device with high-water mark scrolling enabled, you should:</p> <ul style="list-style-type: none"> • Set the <code>scrollPolicy</code> attribute to <code>scroll</code>. <p>For all tables to display on a tablet device with pagination enabled, you should:</p> <ul style="list-style-type: none"> • Set the <code>scrollPolicy</code> attribute to <code>auto</code> (if the page will only run on a tablet) or set to <code>page</code> (if the page may also run on a desktop device). • Set the <code>autoHeightRows</code> attribute to 0. Better, is to set the <code>oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS</code> parameter to <code>auto</code>, as described for geometry management in the first row of this table. <p>If these pagination conditions are not met, the table will display a scroll bar instead of pages.</p> <p>For information about table attributes, see Formatting Tables. For information about flowing layouts and tables, see Geometry Management for the Table, Tree, and Tree Table Components.</p>

E

Quick Start Layout Themes

This appendix shows how each of the quick start layouts are affected when you choose to apply themes to them. ADF Faces provides a number of components that you can use to define the overall layout of a page.

[Figure E-1](#) and [Figure E-2](#) show each of the layouts with and without themes applied. For information about themes, see [Customizing the Appearance Using Styles and Skins](#).

Figure E-1 Quick Start Layouts With and Without Themes

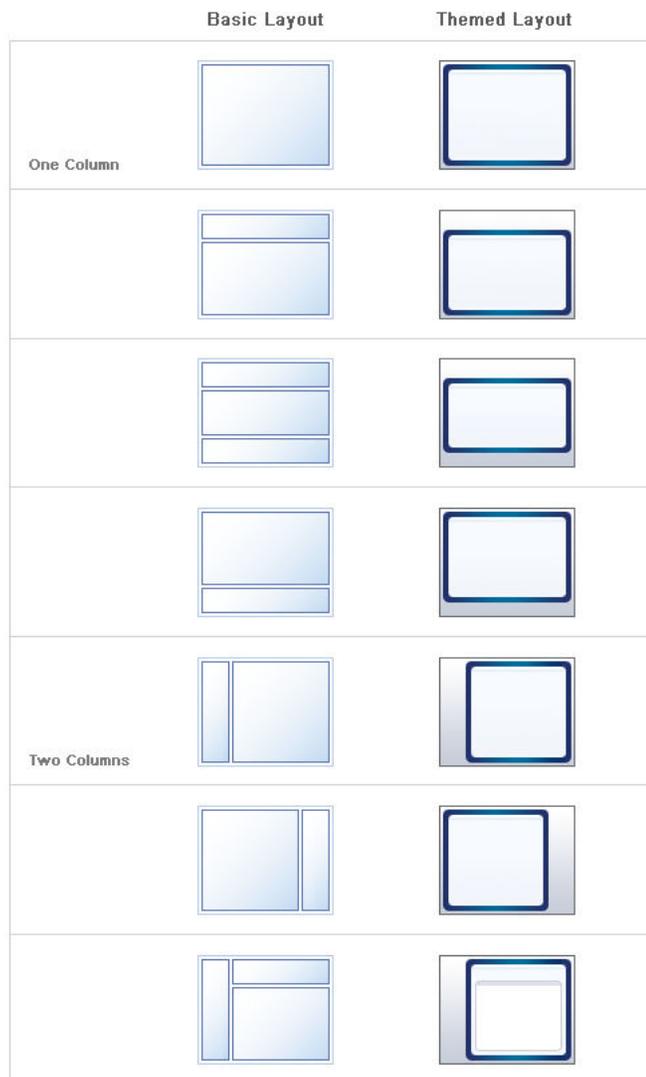


Figure E-2 Quick Start Layouts With and Without Themes

	Basic Layout	Themed Layout
Two Columns		
		
		
		
		
Three Columns		
		
		

F

Code Samples

This appendix provides the full length code samples referenced from sections throughout this guide.

This appendix includes the following sections:

- [Samples for Chapter 4, "Using ADF Faces Client-Side Architecture"](#)
- [Samples for Chapter 29, "Using Map Components"](#)
- [Samples for Chapter 31, "Using Treemap and Sunburst Components"](#)
- [Samples for Chapter 32, "Using Diagram Components"](#)

Samples for Chapter 4, "Using ADF Faces Client-Side Architecture"

The `adf-js-partitions.xml` file is involved in the delivery of JavaScript partitions to the client. It groups the JavaScript features in partitions.

Following are code examples for using ADF Faces architecture

The `adf-js-partitions.xml` File

The default ADF Faces `adf-js-partitions.xml` file has partitions that you can override by creating your own partitions file. See [JavaScript Library Partitioning](#). The example below shows the default ADF Faces `adf-js-partitions.xml` file.

```
<?xml version="1.0" encoding="utf-8"?>

<partitions xmlns="http://xmlns.oracle.com/adf/faces/partition">

  <partition>
    <partition-name>boot</partition-name>
    <feature>AdfBootstrap</feature>
  </partition>

  <partition>
    <partition-name>core</partition-name>

    <feature>AdfCore</feature>

    <!-- Behavioral component super classes -->
    <feature>AdfUIChoose</feature>
    <feature>AdfUICollection</feature>
    <feature>AdfUICommand</feature>
    <feature>AdfUIDialog</feature>
    <feature>AdfUIDocument</feature>
    <feature>AdfUIEditableValue</feature>
    <feature>AdfUIForm</feature>
    <feature>AdfUIGo</feature>
    <feature>AdfUIInput</feature>
```

```

<feature>AdfUIObject</feature>
<feature>AdfUIOutput</feature>
<feature>AdfUIPanel</feature>
<feature>AdfUIPopup</feature>
<feature>AdfUISelectBoolean</feature>
<feature>AdfUISelectInput</feature>
<feature>AdfUISelectOne</feature>
<feature>AdfUISelectMany</feature>
<feature>AdfUIShowDetail</feature>
<feature>AdfUISubform</feature>
<feature>AdfUIValue</feature>

<!-- These are all so common that we group them with core -->
<feature>AdfRichDocument</feature>
<feature>AdfRichForm</feature>
<feature>AdfRichPopup</feature>
<feature>AdfRichSubform</feature>
<feature>AdfRichCommandButton</feature>
<feature>AdfRichCommandLink</feature>

<!--
  Dialog is currently on every page for messaging.  No use
  in putting these in a separate partition.
-->
<feature>AdfRichPanelWindow</feature>
<feature>AdfRichDialog</feature>

<!-- af:showPopupBehavior is so small/common, belongs in core -->
<feature>AdfShowPopupBehavior</feature>
</partition>

<partition>
  <partition-name>accordion</partition-name>
  <feature>AdfRichPanelAccordion</feature>
</partition>

<partition>
  <partition-name>border</partition-name>
  <feature>AdfRichPanelBorderLayout</feature>
</partition>

<partition>
  <partition-name>box</partition-name>
  <feature>AdfRichPanelBox</feature>
</partition>

<partition>
  <partition-name>calendar</partition-name>
  <feature>AdfUICalendar</feature>
  <feature>AdfRichCalendar</feature>
  <feature>AdfCalendarDragSource</feature>
  <feature>AdfCalendarDropTarget</feature>
</partition>

<partition>
  <partition-name>collection</partition-name>
  <feature>AdfUIDecorateCollection</feature>
  <feature>AdfRichPanelCollection</feature>
</partition>

<partition>

```

```
<partition-name>color</partition-name>
<feature>AdfRichChooseColor</feature>
<feature>AdfRichInputColor</feature>
</partition>

<partition>
  <partition-name>date</partition-name>
  <feature>AdfRichChooseDate</feature>
  <feature>AdfRichInputDate</feature>
</partition>

<partition>
  <partition-name>declarativeComponent</partition-name>
  <feature>AdfUIInclude</feature>
  <feature>AdfUIDeclarativeComponent</feature>
  <feature>AdfRichDeclarativeComponent</feature>
</partition>

<partition>
  <partition-name>detail</partition-name>
  <feature>AdfRichShowDetail</feature>
</partition>

<partition>
  <partition-name>dnd</partition-name>
  <feature>AdfDragAndDrop</feature>
  <feature>AdfCollectionDragSource</feature>
  <feature>AdfStampedDropTarget</feature>
  <feature>AdfCollectionDropTarget</feature>
  <feature>AdfAttributeDragSource</feature>
  <feature>AdfAttributeDropTarget</feature>
  <feature>AdfComponentDragSource</feature>
  <feature>AdfDropTarget</feature>
</partition>

<partition>
  <partition-name>detailitem</partition-name>
  <feature>AdfRichShowDetailItem</feature>
</partition>

<partition>
  <partition-name>file</partition-name>
  <feature>AdfRichInputFile</feature>
</partition>

<partition>
  <partition-name>form</partition-name>
  <feature>AdfRichPanelFormLayout</feature>
  <feature>AdfRichPanelLabelAndMessage</feature>
</partition>

<partition>
  <partition-name>format</partition-name>
  <feature>AdfRichOutputFormatted</feature>
</partition>

<partition>
  <partition-name>frame</partition-name>
  <feature>AdfRichInlineFrame</feature>
</partition>
```

```

<partition>
  <partition-name>header</partition-name>
  <feature>AdfRichPanelHeader</feature>
  <feature>AdfRichShowDetailHeader</feature>
</partition>

<partition>
  <partition-name>imagelink</partition-name>
  <feature>AdfRichCommandImageLink</feature>
</partition>

<partition>
  <partition-name>iedit</partition-name>
  <feature>AdfInlineEditing</feature>
</partition>

<partition>
  <partition-name>input</partition-name>
  <feature>AdfRichInputText</feature>
  <feature>AdfInsertTextBehavior</feature>
</partition>

<partition>
  <partition-name>label</partition-name>
  <feature>AdfRichOutputLabel</feature>
</partition>

<partition>
  <partition-name>list</partition-name>
  <feature>AdfRichPanelList</feature>
</partition>

<partition>
  <partition-name>lov</partition-name>
  <feature>AdfUIInputPopup</feature>
  <feature>AdfRichInputComboboxListOfValues</feature>
  <feature>AdfRichInputListOfValues</feature>
</partition>

<partition>
  <partition-name>media</partition-name>
  <feature>AdfRichMedia</feature>
</partition>

<partition>
  <partition-name>message</partition-name>
  <feature>AdfUIMessage</feature>
  <feature>AdfUIMessages</feature>
  <feature>AdfRichMessage</feature>
  <feature>AdfRichMessages</feature>
</partition>

<partition>
  <partition-name>menu</partition-name>
  <feature>AdfRichCommandMenuItem</feature>
  <feature>AdfRichGoMenuItem</feature>
  <feature>AdfRichMenuBar</feature>
  <feature>AdfRichMenu</feature>
</partition>

<partition>

```

```
<partition-name>nav</partition-name>
<feature>AdfUINavigationPath</feature>
<feature>AdfUINavigationLevel</feature>
<feature>AdfRichBreadCrumbs</feature>
<feature>AdfRichCommandNavigationItem</feature>
<feature>AdfRichNavigationPane</feature>
</partition>

<partition>
  <partition-name>note</partition-name>
  <feature>AdfRichNoteWindow</feature>
</partition>

<partition>
  <partition-name>poll</partition-name>
  <feature>AdfUIPoll</feature>
  <feature>AdfRichPoll</feature>
</partition>

<partition>
  <partition-name>progress</partition-name>
  <feature>AdfUIProgress</feature>
  <feature>AdfRichProgressIndicator</feature>
</partition>

<partition>
  <partition-name>print</partition-name>
  <feature>AdfShowPrintablePageBehavior</feature>
</partition>

<partition>
  <partition-name>scrollComponentIntoView</partition-name>
  <feature>AdfScrollComponentIntoViewBehavior</feature>
</partition>

<partition>
  <partition-name>query</partition-name>
  <feature>AdfUIQuery</feature>
  <feature>AdfRichQuery</feature>
  <feature>AdfRichQuickQuery</feature>
</partition>

<partition>
  <partition-name>region</partition-name>
  <feature>AdfUIRegion</feature>
  <feature>AdfRichRegion</feature>
</partition>

<partition>
  <partition-name>reset</partition-name>
  <feature>AdfUIReset</feature>
  <feature>AdfRichResetButton</feature>
</partition>

<partition>
  <partition-name>rte</partition-name>
  <feature>AdfRichTextEditor</feature>
  <feature>AdfRichTextEditorInsertBehavior</feature>
</partition>

<partition>
```

```

    <partition-name>select</partition-name>

    <feature>AdfRichSelectBooleanCheckbox</feature>
    <feature>AdfRichSelectBooleanRadio</feature>
    <feature>AdfRichSelectManyCheckbox</feature>
    <feature>AdfRichSelectOneRadio</feature>
</partition>

<partition>
    <partition-name>selectmanychoice</partition-name>
    <feature>AdfRichSelectManyChoice</feature>
</partition>

<partition>
    <partition-name>selectmanylistbox</partition-name>
    <feature>AdfRichSelectManyListbox</feature>
</partition>

<partition>
    <partition-name>selectonechoice</partition-name>
    <feature>AdfRichSelectOneChoice</feature>
</partition>

<partition>
    <partition-name>selectonelistbox</partition-name>
    <feature>AdfRichSelectOneListbox</feature>
</partition>

<partition>
    <partition-name>shuttle</partition-name>
    <feature>AdfUISelectOrder</feature>
    <feature>AdfRichSelectManyShuttle</feature>
    <feature>AdfRichSelectOrderShuttle</feature>
</partition>

<partition>
    <partition-name>slide</partition-name>
    <feature>AdfRichInputNumberSlider</feature>
    <feature>AdfRichInputRangeSlider</feature>
</partition>

<partition>
    <partition-name>spin</partition-name>
    <feature>AdfRichInputNumberSpinbox</feature>
</partition>

<partition>
    <partition-name>status</partition-name>
    <feature>AdfRichStatusIndicator</feature>
</partition>

<partition>
    <partition-name>stretch</partition-name>
    <feature>AdfRichDecorativeBox</feature>
    <feature>AdfRichPanelSplitter</feature>
    <feature>AdfRichPanelStretchLayout</feature>
    <feature>AdfRichPanelDashboard</feature>
    <feature>AdfPanelDashboardBehavior</feature>
    <feature>AdfDashboardDropTarget</feature>
</partition>

```

```

<partition>
  <partition-name>tabbed</partition-name>
  <feature>AdfUIShowOne</feature>
  <feature>AdfRichPanelTabbed</feature>
</partition>

<partition>
  <partition-name>table</partition-name>
  <feature>AdfUIIterator</feature>
  <feature>AdfUITable</feature>
  <feature>AdfUITable2</feature>
  <feature>AdfUIColumn</feature>
  <feature>AdfRichColumn</feature>
  <feature>AdfRichTable</feature>
</partition>

<partition>
  <partition-name>toolbar</partition-name>
  <feature>AdfRichCommandToolbarButton</feature>
  <feature>AdfRichToolbar</feature>
</partition>

<partition>
  <partition-name>toolbox</partition-name>
  <feature>AdfRichToolbox</feature>
</partition>

<partition>
  <partition-name>train</partition-name>
  <feature>AdfUIProcess</feature>
  <feature>AdfRichCommandTrainStop</feature>
  <feature>AdfRichTrainButtonBar</feature>
  <feature>AdfRichTrain</feature>
</partition>

<partition>
  <partition-name>tree</partition-name>
  <feature>AdfUITree</feature>
  <feature>AdfUITreeTable</feature>
  <feature>AdfRichTree</feature>
  <feature>AdfRichTreeTable</feature>
</partition>

<!--
  Some components which typically do have client-side representation,
  but small enough that we might as well download in a single partition
  in the event that any of these are needed.
-->
<partition>
  <partition-name>uncommon</partition-name>
  <feature>AdfRichGoButton</feature>
  <feature>AdfRichIcon</feature>
  <feature>AdfRichImage</feature>
  <feature>AdfRichOutputText</feature>
  <feature>AdfRichPanelGroupLayout</feature>
  <feature>AdfRichSeparator</feature>
  <feature>AdfRichSpacer</feature>
  <feature>AdfRichGoLink</feature>
</partition>

<partition>

```

```
<partition-name>eum</partition-name>
  <feature>AdfEndUserMonitoring</feature>
</partition>

<partition>
  <partition-name>ads</partition-name>
  <feature>AdfActiveDataService</feature>
</partition>

<partition>
  <partition-name>automation</partition-name>
  <feature>AdfAutomationTest</feature>
</partition>

</partitions>
```

Samples for Chapter 29, "Using Map Components"

Using the `MapProvider` feature, Thematic Map can be configured. The `MapProvider` APIs like `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.MapProvider` and `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.LayerArea` allow the custom basemap to be configured and treated like a built-in basemap with all the same functionalities.

Following are code examples for creating DVT map components.

Sample Code for Thematic Map Custom Base Map

When you create a custom base map for a thematic map you extend `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.MapProvider`. The following is an implementation of the `mapProvider` class for the Canada custom base map example:

```
package oracle.adfdemo.view.feature.rich.thematicMap;

import java.awt.Rectangle;
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import java.net.URL;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.zip.ZipInputStream;

import javax.faces.context.FacesContext;

import oracle.adf.share.logging.ADFLogger;
import oracle.adf.view.faces.bi.component.thematicMap.mapProvider.LayerArea;
```

```
import oracle.adf.view.faces.bi.component.thematicMap.mapProvider.MapProvider;
import
oracle.adf.view.faces.bi.component.thematicMap.mapProvider.utils.MapProviderUtils;

/**
 * Sample MapProvider instance that translates GeoJSON data from a zip file to
 * create a custom Canada basemap.
 */
public class DemoMapProvider extends MapProvider {
    private static String zipFile = "/resources/thematicMap/elocation.zip";
    private static final ADFLogger _logger =
ADFLogger.createADFLogger(DemoMapProvider.class);
    private List<String> hierarchy;

    public DemoMapProvider() {
        hierarchy = new ArrayList<String>();
        hierarchy.add("states");
    }

    @Override
    public Collection<LayerArea> getLayerAreas(String layer, Locale locale) {
        String text = getGeoJsonString();
        Map geoJSON = null;
        try {
            geoJSON = (Map) DemoMapProvider.parseJsonString(text);
        } catch (IOException e) {
            _logger.severe("Could not parse geometries.", e);
        }
        return DemoMapProvider.createLayerAreas(geoJSON);
    }

    @Override
    public Collection<LayerArea> getChildAreas(String layer, Locale locale,
Collection<String> areas) {
        return new ArrayList<LayerArea>();
    }

    @Override
    public double getMaxZoomFactor() {
        return 5.0;
    }

    @Override
    public List<String> getHierarchy() {
        return hierarchy;
    }

    /**
     * Loads the geographic geometries stored in local file and returns as string
     * @return
     */
    private static String getGeoJsonString() {
        StringBuilder sb = new StringBuilder();
        BufferedReader br = null;
        ZipInputStream is = null;
        try {
            URL zipUrl =
FacesContext.getCurrentInstance().getExternalContext().getResource(zipFile);
            is = new ZipInputStream(zipUrl.openStream());
            is.getNextEntry();
            br = new BufferedReader(new InputStreamReader(is, "UTF8"));
        }
    }
}
```

```

String aux = br.readLine();
while (aux != null) {
    sb.append(aux);
    aux = br.readLine();
}
} catch (Exception e) {
    _logger.severe("Could not load geometries from the file " + zipFile, e);
} finally {
    if (br != null) {
        try {
            is.close();
            br.close();
        } catch (IOException e) {
            // ignore
        }
    }
}
return sb.toString();
}

/**
 * Parses a JSON string and converts it to the correct Java object
 * @param str The JSON string to parse
 * @return
 * @throws IOException
 */
private static Object parseJsonString(String str) throws IOException {
    if (str == null)
        return null;

    str = str.trim();

    char firstChar = str.charAt(0);
    if (firstChar == '[' || firstChar == '{') {
        // Array or Map
        // Count the number of open/close arrays or objects
        int numOpen = 0;

        // Quote handling: Count the number of open single/double quotes, except when
        // there is an
        // open one already. This handles nested single quotes in double quotes, and
        // vice versa.
        int numSingleQuotes = 0;
        int numDoubleQuotes = 0;

        // Iterate through and split by pieces
        int prevIndex = 1;
        List<String> parts = new ArrayList<String>();
        for (int i = 1; i < str.length() - 1; i++) {
            char iChar = str.charAt(i);
            if (iChar == '[' || iChar == '{')
                numOpen++;
            else if (iChar == ']' || iChar == '}')
                numOpen--;
            else if (iChar == '\"' && numDoubleQuotes % 2 == 0)
                numSingleQuotes++;
            else if (iChar == '\'' && numSingleQuotes % 2 == 0)
                numDoubleQuotes++;

            // If split index, store the substring
            if (numOpen == 0 && (numSingleQuotes % 2 == 0 && numDoubleQuotes % 2 == 0)

```

```

    && iChar == ',') {
        parts.add(str.substring(prevIndex, i));
        prevIndex = i + 1;
    }
}

// Grab the last part if present
if (prevIndex < str.length() - 1) {
    parts.add(str.substring(prevIndex, str.length() - 1));
}

// Decode the parts into the result
if (firstChar == '[') {
    List ret = new ArrayList();
    for (int arrayIndex = 0; arrayIndex < parts.size(); arrayIndex++)
        ret.add(parseJsonString(parts.get(arrayIndex)));
    return ret;
} else if (firstChar == '{') {
    Map ret = new HashMap();
    for (String part : parts) {
        part = part.trim();
        int colonIndex = part.indexOf(':');
        String mapKey = part.substring(0, colonIndex);
        mapKey = mapKey.substring(1, mapKey.length() - 1); // 1 to -1 to avoid the
quotes
        Object mapValue = parseJsonString(part.substring(colonIndex + 1,
part.length()));
        ret.put(mapKey, mapValue);
    }
    return ret;
}
return null;
} else if (firstChar == '"') // String
    return str.substring(1, str.length() - 1);
else if ("true".equals(str))
    return true;
else if ("false".equals(str))
    return false;
else
    return Double.parseDouble(str);
}

/**
 * Converts a GeoJSON object to a list of LayerArea objects
 * @param geoJSON The GeoJSON object containing this basemap layer's area geometry
data
 * @return
 */
private static List<LayerArea> createLayerAreas(Map geoJSON) {
    List territories = Arrays.asList(MapProviderBean.territoryNames);
    HashMap<String, DemoLayerArea> areaMap = new HashMap<String, DemoLayerArea>();
    if (geoJSON != null) {
        List features = (List) geoJSON.get("features");
        int numFeatures = features.size();
        for (int j = 0; j < numFeatures; j++) {
            Map feature = (Map) features.get(j);
            Map properties = (Map) feature.get("properties");
            String label = (String) properties.get("POLYGON_NAME");
            // We just want to render canada
            if (!territories.contains(label))
                continue;
        }
    }
}

```

```

Map geometry = (Map) feature.get("geometry");

Rectangle2D labelBox = null;
List<Double> labelBoxList = (List<Double>) feature.get("label_box");
if (labelBoxList != null) {
    int minX = (int) (labelBoxList.get(0) / 2000);
    int minY = (int) (labelBoxList.get(1) / 2000);
    int maxX = (int) (labelBoxList.get(2) / 2000);
    int maxY = (int) (labelBoxList.get(3) / 2000);
    labelBox = new Rectangle(minX, -minY, maxX - minX, maxY - minY);
}
DemoLayerArea area = areaMap.get(label);
if (area != null)
    area.setPath(area.getPath() +
DemoMapProvider.simplifyGeometries(geometry));
else
    areaMap.put(label,
                new DemoLayerArea(label, null, label, labelBox,
DemoMapProvider.simplifyGeometries(geometry),
                                null));
}
}

List<LayerArea> layerAreas = new ArrayList<LayerArea>();
for (Map.Entry<String, DemoLayerArea> entry : areaMap.entrySet())
    layerAreas.add(entry.getValue());
return layerAreas;
}

/**
 * Converts and simplifies area geometries to relative path commands
 * @param geometry The map containing an area's coordinates
 * @return
 */
private static String simplifyGeometries(Map geometry) {
    StringBuilder sb = new StringBuilder();
    String type = (String) geometry.get("type");
    List coords = (List) geometry.get("coordinates");
    int len = coords.size();
    if ("Polygon".equals(type)) {
        for (int i = 0; i < len; i++)
            sb.append(DemoMapProvider.simplifyGeometriesHelper((List) coords.get(i)));
    } else if ("MultiPolygon".equals(type)) {
        for (int i = 0; i < len; i++) {
            List nestCoords = (List) coords.get(i);
            int nestCoordsLen = nestCoords.size();
            for (int j = 0; j < nestCoordsLen; j++)
                sb.append(DemoMapProvider.simplifyGeometriesHelper((List) coords.get(j)));
        }
    }
    return sb.toString();
}

/**
 * Helper method for parsing a GeoJSON geometry object
 * @param coords The list of coordinates to simplify and convert to a relative
path command
 * @return
 */
private static String simplifyGeometriesHelper(List coords) {

```

```

        List<Point2D> points = new ArrayList<Point2D>();
        int len = coords.size();
        // Convert coordinates to Point2D objects so we can use MapProviderUtils to
        simplify area geometries
        // Also reduce data precision by dividing by 2000
        for (int i = 0; i < len; i += 2)
            points.add(new Point2D.Double(Math.floor((Double) coords.get(i) / 2000),
                -Math.floor((Double) coords.get(i + 1) / 2000)));
        return MapProviderUtils.convertToPath(points);
    }
}

```

Sample Code for Thematic Map Custom Base Map Area Layer

When you create a custom base map for a thematic map you extend `oracle.adf.view.faces.bi.component.thematicMap.mapProvider.LayerArea` to get the data that the thematic map component requires to render a `dvt:areaLayer`. The following is an implementation of the `layerArea` class for the Canada custom base map example:

```

package oracle.adfdemo.view.feature.rich.thematicMap;

import java.awt.geom.Rectangle2D;

import oracle.adf.view.faces.bi.component.thematicMap.mapProvider.LayerArea;

public class DemoLayerArea extends LayerArea {

    private String id;
    private String shortLabel;
    private String longLabel;
    private Rectangle2D labelBox;
    private String path;
    private String parent;

    public DemoLayerArea(String id, String shortLabel, String longLabel, Rectangle2D
labelBox, String path, String parent) {
        this.id = id;
        this.shortLabel = shortLabel;
        this.longLabel = longLabel;
        this.labelBox = labelBox;
        this.path = path;
        this.parent = parent;
    }

    @Override
    public String getId() {
        return id;
    }

    @Override
    public String getShortLabel() {
        return shortLabel;
    }

    @Override
    public String getLongLabel() {
        return longLabel;
    }
}

```

```

@Override
public Rectangle2D getLabelBox() {
    return labelBox;
}

@Override
public String getPath() {
    return path;
}

public void setPath(String path) {
    this.path = path;
}

@Override
public String getParent() {
    return parent;
}
}

```

Samples for Chapter 31, "Using Treemap and Sunburst Components"

In order to add data to the treemap or sunburst using UI-first development, create the classes, managed beans, and methods that will create the tree model and reference the classes, beans, or methods in JDeveloper.

Following are code examples for creating treemap and sunburst components.

Sample Code for Treemap and Sunburst Census Data Example

When you create a treemap or sunburst using UI-first development, you can use Java classes and managed beans to define the tree node and tree model, populate the tree with data and add additional methods as needed to configure the treemap or sunburst.

The example below shows a code sample defining the tree node in the census data example. Note that the required settings for label, size, and color are passed in as parameters to the tree node.

```

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

public class TreeNode {
    private final String m_text;
    private final Number m_size;
    private final Color m_color;
    private final List<TreeNode> m_children = new ArrayList<TreeNode>();

    public TreeNode(String text, Number size, Color color) {
        m_text = text;
        m_size = size;
        m_color = color;
    }

    public String getText() {
        return m_text;
    }
}

```

```

public Number getSize() {
    return m_size;
}
public Color getColor() {
    return m_color;
}
public void addChild(TreeNode child) {
    m_children.add(child);
}
public void addChildren(List<TreeNode> children) {
    m_children.addAll(children);
}
public List<TreeNode> getChildren() {
    return m_children;
}
@Override
public String toString() {
    return m_text + ": " + m_color + " " + Math.round(m_size.doubleValue());
}
}

```

To supply data to the treemap or sunburst in UI-first development, add a class or managed bean to your application that extends the tree node in the above example and populates it with data. The class to set up the tree model must be an implementation of the `org.apache.myfaces.trinidad.model.TreeModel` class. Once the tree model is defined, create a method that implements the `org.apache.myfaces.trinidad.model.ChildPropertyTreeModel` to complete the tree model.

The example below shows a sample class that sets up the root and child node structure, populates the child levels with data and defines the color and node sizes in the census data example.

```

import java.awt.Color;

import java.util.ArrayList;
import java.util.List;

import org.apache.myfaces.trinidad.model.ChildPropertyTreeModel;
import org.apache.myfaces.trinidad.model.TreeModel;

public class CensusData {

    public static TreeModel getUnitedStatesData() {
        return getModel(ROOT);
    }

    public static TreeModel getRegionWestData() {
        return getModel(REGION_W);
    }

    public static TreeModel getRegionNortheastData() {
        return getModel(REGION_NE);
    }

    public static TreeModel getRegionMidwestData() {
        return getModel(REGION_MW);
    }

    public static TreeModel getRegionSouthData() {

```

```

        return getModel(REGION_S);
    }

    public static TreeModel getDivisionPacificData() {
        return getModel(DIVISION_P);
    }

    private static TreeModel getModel(DataItem rootItem) {
        TreeNode root = getTreeNode(rootItem);
        return new ChildPropertyTreeModel(root, "children");
    }

    private static TreeNode getTreeNode(DataItem dataItem)
    {
        // Create the node itself
        TreeNode node = new CensusTreeNode(dataItem.getName(),
            dataItem.getPopulation(),
            getColor(dataItem.getIncome(), MIN_INCOME, MAX_INCOME),
            dataItem.getIncome());

        // Create its children
        List<TreeNode> children = new ArrayList<TreeNode>();
        for(DataItem childItem : dataItem.children) {
            children.add(getTreeNode(childItem));
        }

        // Add the children and return
        node.addChildren(children);
        return node;
    }

    private static Color getColor(double value, double min, double max) {
        double percent = Math.max((value - min) / max, 0);
        if(percent > 0.5) {
            double modifier = (percent - 0.5) * 2;
            return new Color((int)(modifier*102), (int)(modifier*153), (int)(modifier*51));
        }
        else {
            double modifier = percent * 2;
            return new Color((int)(modifier*204), (int)(modifier*51), 0);
        }
    }

    public static class DataItem {
        private final String name;
        private final int population;
        private final int income;
        private final List<DataItem> children;

        public DataItem(String name, int population, int income) {
            this.name = name;
            this.population = population;
            this.income = income;
            this.children = new ArrayList<DataItem>();
        }

        public void addChild(DataItem child) {
            this.children.add(child);
        }

        public String getName() {

```

```

        return name;
    }

    public int getPopulation() {
        return population;
    }

    public int getIncome() {
        return income;
    }

    public List<CensusData.DataItem> getChildren() {
        return children;
    }
}

private static final int MIN_INCOME = 0;
private static final int MAX_INCOME = 70000;

private static final DataItem ROOT = new DataItem("United States", 301461533,
51425);

private static final DataItem REGION_NE = new DataItem("Northeast Region",
54906297, 57208);
private static final DataItem REGION_MW = new DataItem("Midwest Region", 66336038,
49932);
private static final DataItem REGION_S = new DataItem("South Region", 110450832,
47204);
private static final DataItem REGION_W = new DataItem("West Region", 69768366,
56171);

private static final DataItem DIVISION_NE = new DataItem("New England", 14315257,
61511);
private static final DataItem DIVISION_MA = new DataItem("Middle Atlantic",
40591040, 55726);
private static final DataItem DIVISION_ENC = new DataItem("East North Central",
46277998, 50156);
private static final DataItem DIVISION_WNC = new DataItem("West North Central",
20058040, 49443);
private static final DataItem DIVISION_SA = new DataItem("South Atlantic",
57805475, 50188);
private static final DataItem DIVISION_ESC = new DataItem("East South Central",
17966553, 41130);
private static final DataItem DIVISION_WSC = new DataItem("West South Central",
34678804, 45608);
private static final DataItem DIVISION_M = new DataItem("Mountain", 21303294,
51504);
private static final DataItem DIVISION_P = new DataItem("Pacific", 48465072,
58735);

static {
    // Set up the regions
    ROOT.addChild(REGION_NE);
    ROOT.addChild(REGION_MW);
    ROOT.addChild(REGION_S);
    ROOT.addChild(REGION_W);

    // Set up the divisions
    REGION_NE.addChild(DIVISION_NE);
    REGION_NE.addChild(DIVISION_MA);
    REGION_MW.addChild(DIVISION_ENC);

```

```

REGION_MW.addChild(DIVISION_WNC);
REGION_S.addChild(DIVISION_SA);
REGION_S.addChild(DIVISION_ESC);
REGION_S.addChild(DIVISION_WSC);
REGION_W.addChild(DIVISION_M);
REGION_W.addChild(DIVISION_P);

// Set up the states
DIVISION_NE.addChild(new DataItem("Connecticut", 3494487, 67721));
DIVISION_NE.addChild(new DataItem("Maine", 1316380, 46541));
DIVISION_NE.addChild(new DataItem("Massachusetts", 6511176, 64496));
DIVISION_NE.addChild(new DataItem("New Hampshire", 1315419, 63033));
DIVISION_NE.addChild(new DataItem("Rhode Island", 1057381, 55569));
DIVISION_NE.addChild(new DataItem("Vermont", 620414, 51284));

DIVISION_MA.addChild(new DataItem("New Jersey", 8650548, 68981));
DIVISION_MA.addChild(new DataItem("New York", 19423896, 55233));
DIVISION_MA.addChild(new DataItem("Pennsylvania", 12516596, 49737));

DIVISION_ENC.addChild(new DataItem("Indiana", 6342469, 47465));
DIVISION_ENC.addChild(new DataItem("Illinois", 12785043, 55222));
DIVISION_ENC.addChild(new DataItem("Michigan", 10039208, 48700));
DIVISION_ENC.addChild(new DataItem("Ohio", 11511858, 47144));
DIVISION_ENC.addChild(new DataItem("Wisconsin", 5599420, 51569));

DIVISION_WNC.addChild(new DataItem("Iowa", 2978880, 48052));
DIVISION_WNC.addChild(new DataItem("Kansas", 2777835, 48394));
DIVISION_WNC.addChild(new DataItem("Minnesota", 5188581, 57007));
DIVISION_WNC.addChild(new DataItem("Missouri", 5904382, 46005));
DIVISION_WNC.addChild(new DataItem("Nebraska", 1772124, 47995));
DIVISION_WNC.addChild(new DataItem("North Dakota", 639725, 45140));
DIVISION_WNC.addChild(new DataItem("South Dakota", 796513, 44828));

DIVISION_SA.addChild(new DataItem("Delaware", 863832, 57618));
DIVISION_SA.addChild(new DataItem("District of Columbia", 588433, 56519));
DIVISION_SA.addChild(new DataItem("Florida", 18222420, 47450));
DIVISION_SA.addChild(new DataItem("Georgia", 9497667, 49466));
DIVISION_SA.addChild(new DataItem("Maryland", 5637418, 69475));
DIVISION_SA.addChild(new DataItem("North Carolina", 9045705, 45069));
DIVISION_SA.addChild(new DataItem("South Carolina", 4416867, 43572));
DIVISION_SA.addChild(new DataItem("Virginia", 7721730, 60316));
DIVISION_SA.addChild(new DataItem("West Virginia", 1811403, 37356));

DIVISION_ESC.addChild(new DataItem("Alabama", 4633360, 41216));
DIVISION_ESC.addChild(new DataItem("Kentucky", 4252000, 41197));
DIVISION_ESC.addChild(new DataItem("Mississippi", 2922240, 36796));
DIVISION_ESC.addChild(new DataItem("Tennessee", 6158953, 42943));

DIVISION_WSC.addChild(new DataItem("Arkansas", 2838143, 38542));
DIVISION_WSC.addChild(new DataItem("Louisiana", 4411546, 42167));
DIVISION_WSC.addChild(new DataItem("Oklahoma", 3610073, 41861));
DIVISION_WSC.addChild(new DataItem("Texas", 23819042, 48199));

DIVISION_M.addChild(new DataItem("Arizona", 6324865, 50296));
DIVISION_M.addChild(new DataItem("Colorado", 4843211, 56222));
DIVISION_M.addChild(new DataItem("Idaho", 1492573, 46183));
DIVISION_M.addChild(new DataItem("Montana", 956257, 43089));
DIVISION_M.addChild(new DataItem("Nevada", 2545763, 55585));
DIVISION_M.addChild(new DataItem("New Mexico", 1964860, 42742));
DIVISION_M.addChild(new DataItem("Utah", 2651816, 55642));
DIVISION_M.addChild(new DataItem("Wyoming", 523949, 51990));

```

```

        DIVISION_P.addChild(new DataItem("Alaska", 683142, 64635));
        DIVISION_P.addChild(new DataItem("California", 36308527, 60392));
        DIVISION_P.addChild(new DataItem("Hawaii", 1280241, 64661));
        DIVISION_P.addChild(new DataItem("Oregon", 3727407, 49033));
        DIVISION_P.addChild(new DataItem("Washington", 6465755, 56384));
    }

    public static class CensusTreeNode extends TreeNode {
        private int income;

        public CensusTreeNode(String text, Number size, Color color, int income) {
            super(text, size, color);
            this.income = income;
        }

        public int getIncome() {
            return income;
        }
    }
}

```

Finally, to complete the tree model in UI-first development, add a managed bean to your application that references the class or bean that contains the data and, optionally, add any other methods to customize the treemap or sunburst.

The example below shows a code sample that will instantiate the census treemap and populate it with census data. The example also includes a sample method (`convertToString`) that will convert the treemap node's row data to a string for label display.

```

import org.apache.myfaces.trinidad.component.UIXHierarchy;
import org.apache.myfaces.trinidad.model.RowKeySet;
import org.apache.myfaces.trinidad.model.TreeModel;
import oracle.adf.view.faces.bi.component.treemap.UITreemap;

public class SampleTreemap {
    // Data Model Attrs
    private TreeModel currentModel;
    private final CensusData censusData = new CensusData();
    private String censusRoot = "United States";
    private UITreemap treemap;

    public TreeModel getCensusRootData() {
        return censusData.getUnitedStatesData();
    }

    public TreeModel getCensusData() {
        if ("West Region".equals(censusRoot))
            return censusData.getRegionWestData();
        else if ("South Region".equals(censusRoot))
            return censusData.getRegionSouthData();
        else if ("Midwest Region".equals(censusRoot))
            return censusData.getRegionMidwestData();
        else if ("Northeast Region".equals(censusRoot))
            return censusData.getRegionNortheastData();
        else if ("Pacific Division".equals(censusRoot))
            return censusData.getDivisionPacificData();
        else
            return censusData.getUnitedStatesData();
    }

    public TreeModel getData() {

```

```

    // Return cached data model if available
    if(currentModel != null)
        return currentModel;
    currentModel = getCensusData();
    return currentModel;
}
public void setCensusRoot(String censusRoot) {
    this.censusRoot = censusRoot;
}
public String getCensusRoot() {
    return censusRoot;
}
//Converts the rowKeySet into a string of node text labels.
public static String convertToString(RowKeySet rowKeySet, UIXHierarchy hierarchy) {
    StringBuilder s = new StringBuilder();
    if (rowKeySet != null) {
        for (Object rowKey : rowKeySet) {
            TreeNode rowData = (TreeNode)hierarchy.getRowData(rowKey);
            s.append(rowData.getText()).append(", ");
        }
        // Remove the trailing comma
        if (s.length() > 0)
            s.setLength(s.length() - 2);
    }
    return s.toString();
}
public void setTreemap(UITreemap treemap) {
    this.treemap = treemap;
}
public UITreemap getTreemap() {
    return treemap;
}
}

```

The code to set up the sunburst census sample is nearly identical since both components use the same tree model. See [Code Sample for Sunburst Managed Bean](#) for an example.

Code Sample for Sunburst Managed Bean

The following code sample instantiates the census sunburst and populates it with census data. The example also includes a sample method (`convertToString`) that will convert the sunburst node's row data to a string for label display.

```

import oracle.adf.view.faces.bi.component.sunburst.UISunburst;
import org.apache.myfaces.trinidad.component.UIXHierarchy;
import org.apache.myfaces.trinidad.model.RowKeySet;
import org.apache.myfaces.trinidad.model.TreeModel;

public class SunburstSample {
    // Components
    private UISunburst sunburst;
    // Attributes
    private TreeModel currentModel;
    private final CensusData censusData = new CensusData();
    private String censusRoot = "United States";

    public TreeModel getCensusRootData() {
        return censusData.getUnitedStatesData();
    }
}

```

```

public TreeModel getCensusData() {
    if ("West Region".equals(censusRoot))
        return censusData.getRegionWestData();
    else if ("South Region".equals(censusRoot))
        return censusData.getRegionSouthData();
    else if ("Midwest Region".equals(censusRoot))
        return censusData.getRegionMidwestData();
    else if ("Northeast Region".equals(censusRoot))
        return censusData.getRegionNortheastData();
    else if ("Pacific Division".equals(censusRoot))
        return censusData.getDivisionPacificData();
    else
        return censusData.getUnitedStatesData();
}
public TreeModel getData() {
    // Return cached data model if available
    if(currentModel != null)
        return currentModel;
    currentModel = getCensusData();
    return currentModel;
}
public void setCensusRoot(String censusRoot) {
    this.censusRoot = censusRoot;
}
public String getCensusRoot() {
    return censusRoot;
}
public static String convertToString(RowKeySet rowKeySet, UIXHierarchy hierarchy) {
    StringBuilder s = new StringBuilder();
    if (rowKeySet != null) {
        for (Object rowKey : rowKeySet) {
            TreeNode rowData = (TreeNode)hierarchy.getRowData(rowKey);
            s.append(rowData.getText()).append(", ");
        }
        // Remove the trailing comma
        if (s.length() > 0)
            s.setLength(s.length() - 2);
    }
    return s.toString();
}
public void setSunburst(UISunburst sunburst) {
    this.sunburst = sunburst;
}
public UISunburst getSunburst() {
    return sunburst;
}
}

```

Samples for Chapter 32, "Using Diagram Components"

The Create Diagram wizard in JDeveloper provides a default client layout that you can use or edit to specify the layout of your diagram. The default layout is based on force-directed graph drawing algorithms.

Following are code examples for creating diagram components using the DVT diagram layout framework.

Code Sample for Default Client Layout

When you create a diagram using UI-first development, JDeveloper provides a default client layout option based on force-directed graph drawing algorithms, which can be used and edited.

```

var Application1DiagramLayout = function(layoutContext, optLinkLength, initialTemp) {
    this._layoutContext = layoutContext;
    this._optLinkLength = optLinkLength;
    this._initialTemp = initialTemp;
};

//pad factor for the node size
Application1DiagramLayout.PAD_FACTOR = 1.2;
//initial temperature factor - percent of ideal viewport dimension
Application1DiagramLayout.INIT_TEMP_FACTOR = .25;
//number of iterations to run
Application1DiagramLayout.ITERATIONS = 200;

/**
 * Main function that does the force directed layout (Layout entry point)
 * See algorithm in "Graph Drawing by Force-directed Placement" by Thomas M. J.
Fruchterman and Edward M. Reingold
 * @param {DvtDiagramLayoutContext} layoutContext object that defines a context for
layout call
 */
Application1DiagramLayout.forceDirectedLayout = function(layoutContext)
{
    //pretend that the layout area is just big enough to fit all the nodes
    var maxBounds = Application1DiagramLayout.getMaxNodeBounds(layoutContext);
    var nodeCount = layoutContext.getNodeCount();
    var area = nodeCount * (Application1DiagramLayout.PAD_FACTOR * maxBounds.w) *
(Application1DiagramLayout.PAD_FACTOR * maxBounds.h);
    var initialTemp = Application1DiagramLayout.INIT_TEMP_FACTOR * Math.sqrt(area);

    //optimal link length - default is just the size of an ideal grid cell
    var layoutAttrs = layoutContext.getLayoutAttributes();
    var optLinkLength = (layoutAttrs && layoutAttrs["optimalLinkLength"]) ?
parseFloat(layoutAttrs["optimalLinkLength"]) : Math.sqrt(area / nodeCount);

    //initialize and run the layout
    var layout = new Application1DiagramLayout(layoutContext, optLinkLength,
initialTemp);
    layout.layoutNodes();
    layout.layoutLinks();

    //position labels
    layout.positionNodeLabels();
    layout.positionLinkLabels();
};

/**
 * Layout nodes
 */
Application1DiagramLayout.prototype.layoutNodes = function () {
    this.initForceDirectedPositions();
    var iter = Application1DiagramLayout.ITERATIONS;
    var temp = this._initialTemp;
    for (var i = 0; i < iter; i++) {

```

```

        this.runForceDirectedIteration(temp);
        //after each iteration, decrease the temperature - we do it linearly
        temp = this._initialTemp * (1 - (i + 1)/iter);
    }
};

/**
 * Reposition nodes using force directed algorithm for a single iteration
 * @param {number} t temperature for the iteration that used to limit node
displacement
 */
Application1DiagramLayout.prototype.runForceDirectedIteration = function(t)
{
    //calculate the repulsive force between each two nodes
    var nodeCount = this._layoutContext.getNodeCount();
    for (var ni = 0; ni < nodeCount; ni++) {
        var node = this._layoutContext.getNodeByIndex(ni);
        node.disp = new DvtDiagramPoint(0, 0);
        for (var ni2 = 0; ni2 < nodeCount; ni2++) {
            if (ni == ni2)
                continue;
            var node2 = this._layoutContext.getNodeByIndex(ni2);

            var difference = this._subtractVectors(node.getPosition(),
node2.getPosition());
            var distance = this._vectorLength(difference);
            var repulsion = (this._optLinkLength * this._optLinkLength) / distance;
            node.disp = this._addVectors(node.disp, this._scaleVector(difference,
repulsion / distance ));
        }
    }

    //calculate the attractive force between linked nodes
    var linkCount = this._layoutContext.getLinkCount();
    for (var li = 0; li < linkCount; li++) {
        var link = this._layoutContext.getLinkByIndex(li);
        var node = this._getNodeAtCurrentLevel (link.getStartId());
        var node2 = this._getNodeAtCurrentLevel (link.getEndId());
        if (!node || !node2)
            continue;
        var difference = this._subtractVectors(node.getPosition(), node2.getPosition());
        var distance = this._vectorLength(difference);
        var attraction = (distance * distance) / this._optLinkLength;
        node.disp = this._subtractVectors(node.disp, this._scaleVector(difference,
attraction / distance ));
        node2.disp = this._addVectors(node2.disp, this._scaleVector(difference,
attraction / distance ));
    }

    //limit node displacement by the temperature t and set the position
    for (var ni = 0; ni < nodeCount; ni++) {
        var node = this._layoutContext.getNodeByIndex(ni);
        this._addGravity(node);
        var distance = this._vectorLength(node.disp);
        var pos = this._addVectors(node.getPosition(), this._scaleVector(node.disp,
Math.min(distance, t) / distance));
        node.setPosition(pos);
    }
};

/**

```

```

    * Adds gravity force that attracts a node to the center, the gravity force does not
    allow disconnected nodes and branches to be pushed far away from the center
    * @param {DvtDiagramLayoutContextNode} node object that defines node context for
    the layout
    */
Application1DiagramLayout.prototype._addGravity = function(node) {
    var gravityAdjustment = .2;
    var distance = this._vectorLength(node.getPosition()); //distance from the center
    (0,0)
    var attraction = (distance * distance) / this._optLinkLength;
    node.disp = this._subtractVectors(node.disp,
this._scaleVector(node.getPosition(), attraction / distance * gravityAdjustment ) );
};

/**
 * Initializes node positions - node positions in force directed layout must be
 initialized such that no
 * two nodes have the same position. Position nodes in a circle.
 */
Application1DiagramLayout.prototype.initForceDirectedPositions = function() {
    var nodeCount = this._layoutContext.getNodeCount();
    var angleStep = 2*Math.PI / nodeCount;
    var radius = this._optLinkLength;
    for (var ni = 0; ni < nodeCount; ni++) {
        var x = radius * Math.cos(angleStep * ni);
        var y = radius * Math.sin(angleStep * ni);
        var node = this._layoutContext.getNodeByIndex(ni);
        node.setPosition(new DvtDiagramPoint(x, y));
    }
};

/**
 * Calculate vector length
 * @param {DvtDiagramPoint} p vector
 * @return {number} vector length
 */
Application1DiagramLayout.prototype._vectorLength = function(p) {
    return Math.sqrt(p.x * p.x + p.y * p.y);
};

/**
 * Scale vector
 * @param {DvtDiagramPoint} p vector
 * @param {number} scale scale
 * @return {DvtDiagramPoint} resulting vector
 */
Application1DiagramLayout.prototype._scaleVector = function(p, scale) {
    return new DvtDiagramPoint(p.x * scale, p.y * scale);
};

/**
 * Adds vectors
 * @param {DvtDiagramPoint} p1 vector
 * @param {DvtDiagramPoint} p2 vector
 * @return {DvtDiagramPoint} resulting vector
 */
Application1DiagramLayout.prototype._addVectors = function(p1, p2) {
    return new DvtDiagramPoint(p1.x + p2.x, p1.y + p2.y);
};

/**

```

```

* Subtract vectors
* @param {DvtDiagramPoint} p1 vector
* @param {DvtDiagramPoint} p2 vector
* @return {DvtDiagramPoint} resulting vector
*/
Application1DiagramLayout.prototype._subtractVectors = function(p1, p2) {
    return new DvtDiagramPoint(p1.x - p2.x, p1.y - p2.y);
};

/**
 * Finds a node for the link by the node id. In case of a link that does not connect
 * nodes across containers, that will be a node itself.
 * In case when a link connects nodes across containers, that might be one of the
 * ancestor nodes - the node that has been processed at the current level.
 * @param {string} nodeId id of the node to check
 */
Application1DiagramLayout.prototype._getNodeAtCurrentLevel = function(nodeId) {
    var node;
    do {
        if (!nodeId)
            return null;
        node = this._layoutContext.getNodeById(nodeId);
        nodeId = node.getContainerId();
    } while (!node.disp);
    return node;
};

/**
 * Create links
 */
Application1DiagramLayout.prototype.layoutLinks = function () {
    for (var li = 0; li < this._layoutContext.getLinkCount(); li++) {
        var link = this._layoutContext.getLinkByIndex(li);
        link.setPoints(this.getEndpoints(link));
    }
};

/**
 * Get endpoints for the link
 * @param {DvtDiagramLayoutContextLink} link object that defines link context for
 * the layout
 * @return {array} an array that contains the start X, Y coordinates and the end X,
 * Y coordinates for the link
 */
Application1DiagramLayout.prototype.getEndpoints = function (link) {
    var n1 = this._layoutContext.getNodeById(link.getStartId());
    var n2 = this._layoutContext.getNodeById(link.getEndId());

    var n1Position = n1.getPosition();
    var n2Position = n2.getPosition();
    if (n1.getContainerId() || n2.getContainerId()) { //for cross-container link
        n1Position = this._layoutContext.localToGlobal(new DvtDiagramPoint(0, 0), n1);
        n2Position = this._layoutContext.localToGlobal(new DvtDiagramPoint(0, 0), n2);
    }

    var b1 = n1.getContentBounds();
    var b2 = n2.getContentBounds();

    var startX = n1Position.x + b1.x + .5 * b1.w;
    var startY = n1Position.y + b1.y + .5 * b1.h;
    var endX = n2Position.x + b2.x + .5 * b2.w;

```

```

var endY = n2Position.y + b2.y + .5 * b2.h;

b1 = new DvtDiagramRectangle(n1Position.x + b1.x, n1Position.y + b1.y, b1.w, b1.h);
b2 = new DvtDiagramRectangle(n2Position.x + b2.x, n2Position.y + b2.y, b2.w,
b2.h);
var startP = this._findLinkNodeIntersection(b1, startX, startY, endX, endY,
link.getStartConnectorOffset());
var endP = this._findLinkNodeIntersection(b2, endX, endY, startX, startY,
link.getEndConnectorOffset());
return [startP.x, startP.y, endP.x, endP.y];
};

/**
 * Find a point where a link line intersects the node boundary - use that point as
the start or the end connection point
 * @param {DvtDiagramRectangle} rect the bounds of the node content
 * @param {number} startX x coordinate for the line start
 * @param {number} startY y coordinate for the line start
 * @param {number} endX x coordinate for the line end
 * @param {number} endY y coordinate for the line end
 * @param {number} connOffset the offset of the start connector
 * @return {DvtDiagramPoint} a point where a link line intersects the node boundary
 */
Application1DiagramLayout.prototype._findLinkNodeIntersection = function (rect,
startX, startY, endX, endY, connOffset) {

var lineAngle = Math.atan2(endY - startY, endX - startX);
var cornerAngle = Math.atan2(rect.h, rect.w); // rectangle diagonal from top left
to right bottom
var bottomRightAngle = cornerAngle;
var bottomLeftAngle = Math.PI - bottomRightAngle;
var topRightAngle = - bottomRightAngle;
var topLeftAngle = - bottomLeftAngle;
var x = 0, y = 0;
if (lineAngle >= topLeftAngle && lineAngle <= topRightAngle) { // side top
x = rect.x + rect.w * .5 + Math.tan(Math.PI / 2 - lineAngle) * (-rect.h * .5);
y = rect.y;
}
else if (lineAngle <= bottomLeftAngle && lineAngle >= bottomRightAngle) { //
side bottom
x = rect.x + rect.w * .5 + Math.tan(Math.PI / 2 - lineAngle) * (rect.h * .5);
y = rect.y + rect.h;
}
else if (lineAngle <= bottomRightAngle && lineAngle >= topRightAngle) { // side
right
x = rect.x + rect.w;
y = rect.y + rect.h * .5 + Math.tan(lineAngle) * (rect.w * .5);
}
else { //side left
x = rect.x;
y = rect.y + rect.h * .5 + Math.tan(lineAngle) * (- rect.w * .5);
}

if (connOffset) {
x += Math.cos(lineAngle) * connOffset;
y += Math.sin(lineAngle) * connOffset;
}
return new DvtDiagramPoint(x, y);
};

/**

```

```

    * Center node label below the node
    */
Application1DiagramLayout.prototype.positionNodeLabels = function () {
    for (var ni = 0; ni < this._layoutContext.getNodeCount();ni++) {
        var node = this._layoutContext.getNodeByIndex(ni);
        var nodeLabelBounds = node.getLabelBounds();
        if (nodeLabelBounds) {
            var nodeBounds = node.getContentBounds();
            var nodePos = node.getPosition();
            var labelX = nodeBounds.x + nodePos.x + .5 * (nodeBounds.w -
nodeLabelBounds.w);
            var labelY = nodeBounds.y + nodePos.y + nodeBounds.h;
            node.setLabelPosition(new DvtDiagramPoint(labelX, labelY));
        }
    }
};

/**
 * Position link label at the link center
 */
Application1DiagramLayout.prototype.positionLinkLabels = function (layoutContext) {
    for (var ni = 0;ni < this._layoutContext.getLinkCount();ni++) {
        var link = this._layoutContext.getLinkByIndex(ni);
        var linkLabelBounds = link.getLabelBounds();
        if (linkLabelBounds) {
            var points = link.getPoints();
            if (points.length >=4) {
                var startX = points[0], endX = points[points.length-2];
                var startY = points[1], endY = points[points.length-1];
                var labelX = startX + .5 * (endX - startX - linkLabelBounds.w);
                var labelY = startY + .5 * (endY - startY - linkLabelBounds.h);
                link.setLabelPosition(new DvtDiagramPoint(labelX, labelY));
            }
        }
    }
};

/**
 * Helper function that finds max node width and height
 * @param {DvtDiagramLayoutContext} layoutContext Object that defines a context for
layout call
 * @return {DvtDiagramRectangle} a rectangle that represent a node with max width
and max height
 */
Application1DiagramLayout.getMaxNodeBounds = function (layoutContext) {
    var nodeCount = layoutContext.getNodeCount();
    var maxW = 0 , maxH = 0;
    for (var ni = 0;ni < nodeCount;ni++) {
        var node = layoutContext.getNodeByIndex(ni);
        var bounds = node.getContentBounds();
        maxW = Math.max(bounds.w, maxW);
        maxH = Math.max(bounds.h, maxH);
    }
    return new DvtDiagramRectangle(0, 0, maxW, maxH);
};

```

Code Sample for Simple Circle Layout

For a simple circle layout of nodes and links.

```

var LearningDiagramLayouts = {
    simpleVertical : function (layoutContext) {
        var largestNodeBounds =
LearningDiagramLayouts.getMaxNodeBounds(layoutContext);
        var nodeGap = (2 * largestNodeBounds.h);
        var nodeCount = layoutContext.getNodeCount();
        var linkCount = layoutContext.getLinkCount();
        var xPos = 0;
        var yPos = 0;
        var labelXPos = largestNodeBounds.w + 10;

        for (var i = 0; i < nodeCount; i++) {
            var node = layoutContext.getNodeByIndex(i);
            var nodeHeight = node.getContentBounds().h;
            node.setPosition(new DvtDiagramPoint(xPos, yPos));

            var labelHeight = node.getLabelBounds().h;
            var labelYPos = yPos + ((nodeHeight - labelHeight) / 2);

            node.setLabelPosition(new DvtDiagramPoint(labelXPos, labelYPos));
            yPos += (nodeHeight + nodeGap);
        }

        var linkGap = 4;
        var largestLinkLabelBounds =
LearningDiagramLayouts.getMaxLinkLabelBounds(layoutContext);
        for (var j = 0; j < linkCount; j++) {
            var link = layoutContext.getLinkByIndex(j);
            var startNode = layoutContext.getNodeById(link.getStartId());
            var endNode = layoutContext.getNodeById(link.getEndId());
            var linkStartPos = LearningDiagramLayouts.getNodeEdgeMidPoint(startNode,
"s");
            var linkEndPos = LearningDiagramLayouts.getNodeEdgeMidPoint(endNode,
"n");

            var linkLabelXPos;
            var linkLabelYPos = linkStartPos.y + ((linkEndPos.y - linkStartPos.y -
link.getLabelBounds().h) / 2);

            if (linkEndPos.y > linkStartPos.y) {
                link.setPoints([linkStartPos.x, (linkStartPos.y + linkGap),
linkEndPos.x, (linkEndPos.y - linkGap)]);
                linkLabelXPos = linkStartPos.x - (link.getLabelBounds().w + 20);
            } else {
                // need to re-route in this case
                // Segment 1 - keep heading down for a short distance
                var turn1 = new DvtDiagramPoint();
                turn1.x = linkStartPos.x;
                turn1.y = linkStartPos.y + (largestNodeBounds.h/2);
                // Segment 2 - head left far enough to avoid overlap with the other
link labels
                var turn2 = new DvtDiagramPoint();
                turn2.x = turn1.x - (largestLinkLabelBounds.w + 40);
                turn2.y = turn1.y;
                // Segment 3 - Back up the diagram to a point equally above the end
node
                var turn3 = new DvtDiagramPoint();
                turn3.x = turn2.x;
                turn3.y = linkEndPos.y - (largestNodeBounds.h/2);
                // Segment 4 - Back to the center line
                var turn4 = new DvtDiagramPoint();

```

```

        turn4.x = linkEndPos.x;
        turn4.y = turn3.y;
        // Segment 5 - And down to the end node for which
        // we already have the coordinates

        //Now pass in the path:
        link.setPoints([linkStartPos.x, (linkStartPos.y + linkGap),
                       turn1.x, turn1.y,
                       turn2.x, turn2.y,
                       turn3.x, turn3.y,
                       turn4.x, turn4.y,
                       linkEndPos.x, (linkEndPos.y -
linkGap)]);

        //Finally work out the X position of the label in this case
        linkLabelXPos = turn3.x - (link.getLabelBounds().w + 20);
    }

    link.setLabelPosition(new DvtDiagramPoint(linkLabelXPos, linkLabelYPos));
}
/**
 * Utility function to return the size information for
 * the largest node being handled by the layout
 */
getMaxNodeBounds : function (layoutContext) {
    var nodeCount = layoutContext.getNodeCount();
    var maxW = 0;
    var maxH = 0;
    for (var i = 0; i < nodeCount; i++) {
        var node = layoutContext.getNodeByIndex(i);
        var bounds = node.getContentBounds();
        maxW = Math.max(bounds.w, maxW);
        maxH = Math.max(bounds.h, maxH);
    }
    return new DvtDiagramRectangle(0, 0, maxW, maxH);
},
/**
 * Utility function to return the size information for
 * the largest link label being handled by the layout
 */
getMaxLinkLabelBounds : function (layoutContext) {
    var linkCount = layoutContext.getLinkCount();
    var maxW = 0;
    var maxH = 0;
    for (var i = 0; i < linkCount; i++) {
        var link = layoutContext.getLinkByIndex(i);
        var bounds = link.getLabelBounds();
        maxW = Math.max(bounds.w, maxW);
        maxH = Math.max(bounds.h, maxH);
    }
    return new DvtDiagramRectangle(0, 0, maxW, maxH);
},
/**
 * Utility function to return the midpoint of a node edge.
 * Edges are identified as compass points n e s w
 */
getNodeEdgeMidPoint : function (node, edge) {
    var nodeSize = node.getContentBounds();
    var nodePosition = node.getPosition();

```

```

var xpos;
var ypos;

switch (edge) {
  case "n":
  case "s":
    xpos = nodePosition.x + (nodeSize.w / 2);
    break;
  case "e":
    xpos = nodePosition.x + nodeSize.w;
    break;
  case "w":
    xpos = nodePosition.x;
    break;
  default :
    xpos = 0;
}

switch (edge) {
  case "e":
  case "w":
    ypos = nodePosition.y + (nodeSize.h / 2);
    break;
  case "s":
    ypos = nodePosition.y + nodeSize.h;
    break;
  case "n":
    ypos = nodePosition.y;
    break;
  default :
    ypos = 0;
}

return new DvtDiagramPoint(xpos, ypos);
}
}

```

Code Sample for Simple Vertical Layout

This diagram layout uses a restricted vertical layout with the assumption of no more than one incoming and one outgoing link per node.

```

var LearningDiagramLayouts = {
  simpleVertical : function (layoutContext) {
    var largestNodeBounds =
LearningDiagramLayouts.getMaxNodeBounds(layoutContext);
    var nodeGap = (2 * largestNodeBounds.h);
    var nodeCount = layoutContext.getNodeCount();
    var linkCount = layoutContext.getLinkCount();
    var xPos = 0;
    var yPos = 0;
    var labelXPos = largestNodeBounds.w + 10;

    for (var i = 0; i < nodeCount; i++) {
      var node = layoutContext.getNodeByIndex(i);
      var nodeHeight = node.getContentBounds().h;
      node.setPosition(new DvtDiagramPoint(xPos, yPos));

      var labelHeight = node.getLabelBounds().h;
      var labelYPos = yPos + ((nodeHeight - labelHeight) / 2);

```

```

        node.setLabelPosition(new DvtDiagramPoint(labelXPos, labelYPos));
        yPos += (nodeHeight + nodeGap);
    }

    var linkGap = 4;
    var largestLinkLabelBounds =
LearningDiagramLayouts.getMaxLinkLabelBounds(layoutContext);
    for (var j = 0; j < linkCount; j++) {
        var link = layoutContext.getLinkByIndex(j);
        var startNode = layoutContext.getNodeById(link.getStartId());
        var endNode = layoutContext.getNodeById(link.getEndId());
        var linkStartPos = LearningDiagramLayouts.getNodeEdgeMidPoint(startNode,
"s");
        var linkEndPos = LearningDiagramLayouts.getNodeEdgeMidPoint(endNode,
"n");

        var linkLabelXPos;
        var linkLabelYPos = linkStartPos.y + ((linkEndPos.y - linkStartPos.y -
link.getLabelBounds().h) / 2);

        if (linkEndPos.y > linkStartPos.y) {
            link.setPoints([linkStartPos.x, (linkStartPos.y + linkGap),
linkEndPos.x, (linkEndPos.y - linkGap)]);
            linkLabelXPos = linkStartPos.x - (link.getLabelBounds().w + 20);
        } else {
            // need to re-route in this case
            // Segment 1 - keep heading down for a short distance
            var turn1 = new DvtDiagramPoint();
            turn1.x = linkStartPos.x;
            turn1.y = linkStartPos.y + (largestNodeBounds.h/2);
            // Segment 2 - head left far enough to avoid overlap with the other
link labels
            var turn2 = new DvtDiagramPoint();
            turn2.x = turn1.x - (largestLinkLabelBounds.w + 40);
            turn2.y = turn1.y;
            // Segment 3 - Back up the diagram to a point equally above the end
node
            var turn3 = new DvtDiagramPoint();
            turn3.x = turn2.x;
            turn3.y = linkEndPos.y - (largestNodeBounds.h/2);
            // Segment 4 - Back to the center line
            var turn4 = new DvtDiagramPoint();
            turn4.x = linkEndPos.x;
            turn4.y = turn3.y;
            // Segment 5 - And down to the end node for which
            // we already have the coordinates

            //Now pass in the path:
            link.setPoints([linkStartPos.x, (linkStartPos.y + linkGap),
                turn1.x, turn1.y,
                turn2.x, turn2.y,
                turn3.x, turn3.y,
                turn4.x, turn4.y,
                linkEndPos.x, (linkEndPos.y -
linkGap)]);

            //Finally work out the X position of the label in this case
            linkLabelXPos = turn3.x - (link.getLabelBounds().w + 20);
        }
    }

```

```

        link.setLabelPosition(new DvtDiagramPoint(linkLabelXPos, linkLabelYPos));
    }
},
/**
 * Utility function to return the size information for
 * the largest node being handled by the layout
 */
getMaxNodeBounds : function (layoutContext) {
    var nodeCount = layoutContext.getNodeCount();
    var maxW = 0;
    var maxH = 0;
    for (var i = 0; i < nodeCount; i++) {
        var node = layoutContext.getNodeByIndex(i);
        var bounds = node.getContentBounds();
        maxW = Math.max(bounds.w, maxW);
        maxH = Math.max(bounds.h, maxH);
    }
    return new DvtDiagramRectangle(0, 0, maxW, maxH);
},
/**
 * Utility function to return the size information for
 * the largest link label being handled by the layout
 */
getMaxLinkLabelBounds : function (layoutContext) {
    var linkCount = layoutContext.getLinkCount();
    var maxW = 0;
    var maxH = 0;
    for (var i = 0; i < linkCount; i++) {
        var link = layoutContext.getLinkByIndex(i);
        var bounds = link.getLabelBounds();
        maxW = Math.max(bounds.w, maxW);
        maxH = Math.max(bounds.h, maxH);
    }
    return new DvtDiagramRectangle(0, 0, maxW, maxH);
},
/**
 * Utility function to return the midpoint of a node edge.
 * Edges are identified as compass points n e s w
 */
getNodeEdgeMidPoint : function (node, edge) {
    var nodeSize = node.getContentBounds();
    var nodePosition = node.getPosition();
    var xpos;
    var ypos;

    switch (edge) {
        case "n":
        case "s":
            xpos = nodePosition.x + (nodeSize.w / 2);
            break;
        case "e":
            xpos = nodePosition.x + nodeSize.w;
            break;
        case "w":
            xpos = nodePosition.x;
            break;
        default :
            xpos = 0;
    }
}

```

```
switch (edge) {  
    case "e":  
    case "w":  
        ypos = nodePosition.y + (nodeSize.h / 2);  
        break;  
    case "s":  
        ypos = nodePosition.y + nodeSize.h;  
        break;  
    case "n":  
        ypos = nodePosition.y;  
        break;  
    default :  
        ypos = 0;  
}  
  
return new DvtDiagramPoint(xpos, ypos);  
}  
}
```

G

Troubleshooting ADF Faces

This appendix describes common problems that you might encounter when designing the application user interface with the ADF Faces framework and ADF Faces components and explains how to solve them.

This appendix includes the following sections:

- [About Troubleshooting ADF Faces](#)
- [Getting Started with Troubleshooting the View Layer of an ADF Application](#)
- [Using Test Automation for ADF Faces](#)
- [Resolving Common Problems](#)
- [Using My Oracle Support for Additional Troubleshooting Information](#)

In addition to this chapter, review *Error Messages* for information about the error messages you may encounter.

About Troubleshooting ADF Faces

While designing the ADF application user interface, you might face problems. Certain guidelines and processes should be followed to identify, diagnose and resolve problems proactively.

This section provides guidelines and a process for using the information in this appendix. Using the following guidelines and process will focus and minimize the time you spend resolving problems.

Guidelines

When using the information in this chapter, please keep the following best practices in mind:

- After performing any of the solution procedures in this chapter, immediately retry the failed task that led you to this troubleshooting information. If the task still fails when you retry it, perform a different solution procedure in this chapter and then try the failed task again. Repeat this process until you resolve the problem.
- Make notes about the solution procedures you perform, symptoms you see, and data you collect while troubleshooting. If you cannot resolve the problem using the information in this chapter and you must log a service request, the notes you take will expedite the process of solving the problem.

Process

Follow the process outlined in [Table G-1](#) when using the information in this chapter. If the information in a particular section does not resolve your problem, proceed to the next step in this process.

Table G-1 Process for Using the Information in this Chapter

Step	Section to Use	Purpose
1	Getting Started with Troubleshooting the View Layer of an ADF Application	Get started troubleshooting the view layer of an ADF application. The procedures in this section quickly address a wide variety of problems.
2	Resolving Common Problems	Perform problem-specific troubleshooting procedures for the view layer of an ADF application. This section describes: <ul style="list-style-type: none"> • Possible causes of the problems • Solution procedures corresponding to each of the possible causes
3	Using My Oracle Support for Additional Troubleshooting Information	Use My Oracle Support to get additional troubleshooting information. The My Oracle Support web site provides access to several useful troubleshooting resources, including links to Knowledge Base articles, Community Forums, and Discussion pages.
4	Using My Oracle Support for Additional Troubleshooting Information	Log a service request if the information in this chapter and My Oracle Support does not resolve your problem. You can log a service request using My Oracle Support at https://support.oracle.com .

Getting Started with Troubleshooting the View Layer of an ADF Application

Builtin error messages in Oracle ADF alerts users of a problem and also identifies the layer of the application that is causing the problem. To optimizing ADF Faces troubleshooting, certain configuration options should be set.

Oracle ADF has builtin error messages that enable you to determine which layer of your application may be causing a problem. Error messages are the starting point for troubleshooting. You can look up error messages in *Error Messages*, and you may research a particular error message on the web. Error messages that originate from your ADF Business Components model layer will have a JBO prefix, whereas all other ADF layer components, including the ADF Face view layer, will appear as a Java error message with an Oracle package.

Once you are able to identify the layer, you may run diagnostic tools. You may also view log files for recorded errors. You can search the technical forums on Oracle Technology Network for discussions related to an error message. Each of the component layers for Oracle ADF has its own dedicated forum. You can access the forum home page for JDeveloper and Oracle ADF under the Development Tools list on Oracle Technology Network at <https://forums.oracle.com/forums/main.jspx?categoryID=84>.

Before you begin troubleshooting, you should configure the ADF application to make finding and detecting errors easier. [Table G-2](#) summarizes the settings that you can follow to configure the view layer of an ADF application for troubleshooting.

Table G-2 Configuration Options for Optimizing ADF Faces Troubleshooting

Configuration Recommendation	Description
Enable debug output.	<p>To enable debug output, set the following in the <code>trinidad-config.xml</code> file:</p> <pre><adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config"> <debug-output>true</debug-output> <skin-family>oracle</skin-family> </adf-faces-config></pre> <p>Improves the readability of HTML markup in the web browser:</p> <ul style="list-style-type: none"> • Line wraps and indents the output. • Detects and highlights unbalanced elements and other common HTML errors, such as unbalanced elements. • Adds comments that help you to identify which ADF Faces component generated each block of HTML in the browser page.
Disable content compression.	<p>To disable content compression, set the following in the <code>web.xml</code> file:</p> <pre><context-param> <param-name> org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION </param-name> <param-value>true</param-value> </context-param></pre> <p>Improves readability by forcing the use of original uncompressed styles. Unless content compression is disabled, CSS style names and styles will appear compressed and may be more difficult to read. For information about how disabling content compression affects readability, see the "Applying the Finished ADF Skin to Your Web Application" in <i>Developing ADF Skins</i>.</p>
Disable JavaScript compression.	<p>To disable JavaScript compression, set the following in the <code>web.xml</code> file:</p> <pre><context-param> <param-name> org.apache.myfaces.trinidad.DEBUG_JAVA_SCRIPT </param-name> <param-value>true</param-value> </context-param></pre> <p>Allows normally obfuscated JavaScript to appear uncompressed as the source.</p>
Enable client side asserts.	<p>To enable client side asserts, set the following in the <code>web.xml</code> file:</p> <pre><context-param> <param-name> oracle.adf.view.rich.ASSERT_ENABLED </param-name> <param-value>true</param-value> </context-param></pre> <p>Allows warnings of unexpected conditions to be output to the browser console.</p>

Table G-2 (Cont.) Configuration Options for Optimizing ADF Faces Troubleshooting

Configuration Recommendation	Description
Enable client side logging.	<p>To enable client side logging, set the following in the <code>web.xml</code> file:</p> <pre><context-param> <param-name> oracle.adf.view.rich.ASSERT_ENABLED </param-name> <param-value>true</param-value> </context-param></pre> <p>Allows log messages to be output to the browser console. Unless client side logging is enabled, log messages will not be reported in the client.</p>
Enable more detailed server side logging.	<p>To enable more detailed server side logging, shut down the application server and then enter the following setting in the <code>logging.xml</code> file, and restart the server:</p> <pre><logger name="oracle.adf.faces" level="CONFIG" /></pre> <p>or</p> <p>Use the WLST command:</p> <pre>setLogLevel(logger="oracle.adf" level="CONFIG", addLogger=1)</pre> <p>or</p> <p>In Oracle Enterprise Manager Fusion Middleware Control, use the Configuration page to set <code>oracle.adf</code>, <code>oracle.adfinternal</code>, and <code>oracle.jbo</code> to level CONFIG.</p> <p>Allows more detailed log messages to be output to the browser console. Unless server side logging is configured with a log level of CONFIG or higher, useful diagnostic messages may go unreported. Allowed log level settings are: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL. Oracle recommends CONFIG level or higher; the default is SEVERE.</p>
Disable HTTP cache headers.	<p>To disable HTTP cache headers, set the following in the <code>web.xml</code> file:</p> <pre><context-param> <param-name> org.apache.myfaces.trinidad.resource.DEBUG </param-name> <param-value>true</param-value> </context-param></pre> <p>Forces reloading of patched resources. Unless HTTP cache headers are disabled, the browser will cache resources to ensure fast access to resources. After changing the setting, clear the browser cache to force it to reload resources.</p>

Using Test Automation for ADF Faces

It is important to test that the user interface of your ADF application performs as expected. There are tools that can help you achieve such automation of your UI testing to increase the effectiveness, efficiency and coverage of your UI testing.

This section describes best practices for using the ADF Faces test automation framework.

- [Enabling Test Automation for ADF Faces](#)
- [Simulating Mouse Events in ADF Faces Test Automation](#)

Enabling Test Automation for ADF Faces

You can enable test automation by setting the `oracle.adf.view.rich.automation.ENABLED` parameter in the application's `web.xml` file. This parameter notifies ADF Faces that test automation is being used and turns on external component `id` attributes. Note that this parameter only enables the infrastructure for test automation; it does not initiate testing itself, which requires a tool such as the open source Selenium IDE.

Tests that you write that rely on the internal component `id` are always guaranteed to have a value. The disadvantage is that the internal component `id` values may change between ADF Faces releases to support framework enhancements. Therefore, the tests that you write must not rely on the component `id`. To build more robust automated tests, rely on absolute locator `id` attributes and check that these attributes have a value.

The `oracle.adf.view.rich.automation.ENABLED` parameter accepts the following values:

- **OFF**: Indicates the test automation is disabled. This is the appropriate setting for a production environment.
- **FULL**: Notifies ADF Faces that test automation is being used and turns on external component `id` attributes. Use this setting with caution, and only in a protected testing environment.

Note:

When the test automation context parameter is set to `FULL`, the `oracle.adf.view.rich.security.FRAME_BUSTING` context parameter behaves as though it were set to `never`. The security consequence of disabling framebusting is that pages of your application will become vulnerable to clickjacking from malicious websites. For this reason, restrict the use of the test automation to development or staging environments and never enable test automation in a production environment. See [Framebusting](#).

- **SAFE**: Notifies ADF Faces that test automation is being used and turns on external component `id` attributes. This setting is the same as **FULL**, with the following differences in how it treats secure information.

- It does not disable framebusting. That is, the `oracle.adf.view.rich.security.FRAME_BUSTING` context parameter does not behave as though it were set to never.
- `JSESSIONID` will be encrypted.
- Version number information, described in [Version Number Information](#) will not be shown.
- Logging to the server is disabled.

 **Performance Tip:**

When you enable test automation, a client component is created for every component on the page. Therefore, set this parameter to `OFF` in a production environment.

The `oracle.adf.view.rich.automation.ENABLED` context parameter that you define in the application's `web.xml` file is EL bindable. For example, you might want the application to programmatically set test automation mode for a single test user, for all users of the application, or for a particular user group. The following example shows how to use an EL expression to set test automation mode for a test user.

```
<context-param>
<param-name>oracle.adf.view.rich.automation.ENABLED</param-name>
  <param-value>
    #{securityContext.userName == 'testuser' ? 'SAFE' : 'OFF'}
  </param-value>
</context-param>
```

If you enable automation and set `oracle.adf.view.rich.automation.ENABLED` to `SAFE` or `FULL`, a response cookie is enabled automatically. This response cookie indicates that the request is coming from an automation run that allows the server to limit the incoming automation traffic.

Enabling test automation also enables assertions in the running application. If your application exhibits unexpected component behavior and you begin to see new assertion failed errors, you will need to examine the implementation details of your application components. For example, it is not uncommon to discover issues related to popup dialogs, such as user actions that are no longer responded to.

Here are known coding errors that will produce assertion failed errors only after test automation is enabled:

- Your component references an ADF iterator binding that no longer exists in the page definition file. When assertions are not enabled, this error is silent and the component referencing the missing iterator simply does not render.
- Your component is a partial trigger component that is defined not to render (has the attribute setting `rendered="false"`). For example, this use of the `rendered` attribute causes an assertion failed error:

```
<af:button id="hiddenBtn" rendered="false" text="Test"/>
<af:table var="row" id="t1" partialTriggers="::hiddenBtn">
```

The workaround for this error is to use the attribute setting `visible="false"` and not `rendered="false"`.

- Your components were formed with a nesting hierarchy that prevents events from reaching the proper component handlers. For example, this nesting is incorrect:

```
<af:commandLink
  <af:showPopupBehavior
    <af:image
      <af:clientListener
```

and should be rewritten as:

```
<af:commandLink
  <af:image
    <af:showPopupBehavior
  <af:clientListener
```

Note:

System administrators can enable test automation at the level of standalone Oracle WebLogic Server by starting the server with the command line flag - `Doracle.adf.view.rich.automation.ENABLED= automation-mode`. Running your application in an application server instance with test automation enabled overrides the `web.xml` file context parameter setting of the deployed application.

Simulating Mouse Events in ADF Faces Test Automation

During test automation, you can simulate mouse events that you can use to test mouse events. To simulate mouse events, you need to first add a test harness that you can use to set up the environment for a test simulation mouse click before the test-engine mouse-click simulation occurs. The test harness calls the following function.

```
AdfPage.prototype.simulateMouse = function(locator, mouseEvent)
```

Before you call the function, you need to update your test engine to call the API.

The sample API is given below:

```
/**
 * Simulate mouse type event using locator.
 * @param {String} locator locator string in the format <scopedId>#<subid>
 * @param {Object} javascript object containing a sparse set of mouse
event name value pairs that will
 * be used to create a mouse event. Defaults will be used
for any missing fields.
 * The defaults are as follows:
 * type: "click"
 * canBubble: true
 * cancelable:true
 * view: the current window
 * detail: 1
 * screenX: 0
```

```

*           screenY: 0
*           clientX: the client location of the X coordinate
of the center of the DOM
*           element returned by the locator
*           clientY: the client location of the Y coordinate
of the center of the DOM
*           element returned by the locator
*           ctrlKey: false
*           altKey: false
*           shiftKey: false
*           metaKey: false
*           button: 0
*           relatedTarget: null
*           If no mouseEvent is provided, a mouseEvent with all of
the defaults will be used.
* @abstract
*/

```

After you prepare the environment for the mouse event, this method defers to a test engine class instance that you will define. By doing so, you are pointing the test engine class instance back to your normal test engine to test the mouse event. That class continues to do the normal test procedure for a mouse event that you did before the start up of the mouse event.

Now create a test engine class to be an instance of the new abstract API class. Your test engine implementation needs to override this method:

```

AdfDhtmlTestEngine.prototype.simulateDomMouse = function(domElement,
mouseEvent)

```

Following is an example of an instance of `AdfDhtmlTestEngine`, where you need to add the call to the test framework mouse event code that you just created.

```

/**
 * Used for testing a custom implementation of AdfDhtmlTestEngine.
 */
function CustomTestEngine()
{
}

// make CustomTestEngine a subclass of AdfDhtmlTestEngine
AdfObject.createSubclass(CustomTestEngine, AdfDhtmlTestEngine);

/**
 * Simulate mouse event on a dom element.
 * @param {String} domElement the domElement receiving the mouse event
 * @param {Object} javascript object containing a sparse set of mouse
event name value pairs.
 * @Override
 */
CustomTestEngine.prototype.simulateDomMouse = function(domElement,
mouseEvent)
{
// Here, call your custom test framework to actually do the mouse event,

```

```
now that the environment is all ready for this event.  
}
```

Finally, ADF should be aware of the new test engine class. You can specify this using the `web.xml` parameter. This is important for the functioning of test setup code, like `AdfPage.PAGE.simulateMouse()`, that needs to invoke the testing framework function after doing any necessary setup work.

```
<context-param>  
<description>  
This parameter specifies the automation test engine that is being used.  
</description>  
param-name>oracle.adf.view.automation.TEST_ENGINE</param-name>  
<param-value>CustomTestEngine</param-value>
```

Resolving Common Problems

There are some common problems that may arise while developing your ADF application user interface. The list of problems with their causes and possible fixes will help you to resolve issues.

This section describes common problems and solutions.

Application Displays an Unexpected White Background

The ADF application has a default skin that displays a simple or minimal look and feel. The background of the default skin will appear white.

Cause

The skin JAR files did not get deployed correctly to all applications.

Solution

To resolve this problem:

1. Check that the skin JAR files have been deployed to all applications.
2. Check that the skin name is not misspelled in the profile options. See [Applying the Finished ADF Skin to Your Web Application](#) in *Developing ADF Skins*.

Application is Missing Expected Images

The skin application must be packaged as an ADF Library JAR file that includes the image files.

Cause

The skin JAR files were not packaged correctly.

Solution

To resolve this problem:

1. Check that the correct target application version was specified when creating the skin application.
2. Repackage the skin application and create a new ADF Library JAR file, for information on this, see *Packaging an ADF Skin into an ADF Library JAR in Developing ADF Skins*.

ADF Skin Does Not Render Properly

The ADF skin that you created defines style properties that do not render in the browser as expected.

Cause

Not all ADF skin selectors may be customized and customizing non-valid selectors may result in unexpected or inconsistent behavior for your application.

Solution

To resolve this problem:

1. Check the setting of the context initialization parameter in the `web.xml` file. Not all changes that you make to an ADF skin appear immediately if you set the `CHECK_FILE_MODIFICATION` parameter to `true`. You must restart the web application to view changes that you make to icon and ADF skin properties.
2. Check that you customized only valid ADF skin selectors, pseudo-elements, and pseudo-classes, as described in the skinning sections of the ADF Faces component and ADF Data Visualization (DVT) component chapters of this guide.

ADF Data Visualization Components Fail to Display as Expected

Various ADF Data Visualization (DVT) components rely on Flash to display correctly and unless Flash is supported by the platform and browser, your application may not display visual aspects of the DVT components.

Cause

Not all platforms and browsers support Flash. This will force the application to downgrade to the best available fallback. If the platform is not supported, the application displays according to the `flash-player-usage` setting in the `adf-config.xml` file.

Solution

To resolve this problem, reinstall the latest Flash version available for your browser.

High Availability Application Displays a `NotSerializableException`

When you design an application to run in a clustered environment, you must ensure that all managed beans with a life span longer than one request are serializable.

Cause

When the Fusion web application runs in a clustered environment, a portion of the application's state is serialized and copied to another server or a data store at the end of each request so that the state is available to other servers in the cluster.

Specifically, beans stored in session scope, page flow scope, and view scope must be serializable (that is, they implement the `java.io.Serializable` interface).

Solution

To resolve this problem:

1. Enable server checking to ensure no unserializable state content on session attributes is detected. This check is disabled by default to reduce runtime overhead. Serialization checking is supported by the system property `org.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION`. The following are system properties and you must specify them when you start the application server.
2. For high availability testing, start off by validating that the Session and JSF state is serializable by launching the application server with the system property:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=session,tree
```

3. Add the beans option to check that any serializable object in the appropriate map has been marked as dirty if the serialized content of the object has changed during the request:

```
-  
Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=session,tree,beans
```

4. If a JSF state serialization failure is detected, relaunch the application server with the system property to enable component and property flags and rerun the test:

```
-Dorg.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION=all
```

Unable to Reproduce Problem in All Web Browsers

You run the application in Microsoft Windows Internet Explorer and verify a problem but when you run the application in Mozilla Firefox, the problem does not reproduce. These problems are often visual in nature, such as unintended extra space separating areas within a web page.

Cause

Settings between browsers vary and can lead to differences in the visual appearance of your application.

Solution

To resolve this problem:

1. Check browser security settings to ensure they are not misconfigured. For example, confirm that you have not disabled JavaScript, XML HTTP, or popups.
2. Confirm that Internet Explorer is not being run in compatibility mode. If you see a dialog that states "the current compatibility setting is not supported," disable compatibility mode in the browser's Tools menu.
3. If you observe a JavaScript error, then it is most likely a bug in the browser. However, it could be an ADF Faces-specific JavaScript error.
4. Optionally, configure an `@agent` at-rule in the ADF skin file of the application to define an appearance specific to the desired browser. See "Working with At-Rules" in *Developing ADF Skins*.

Application is Missing Content

The application pages may display areas that appear empty where content is expected.

Cause

The cause depends on the application design. For example, authorization that you enforce in the application may be unintentionally preventing the application from displaying content. Or, when portlets are used, the portlet server may be down.

Solution

To resolve this problem:

1. Check the log file for exceptions. Oracle recommends changing the log level to a lower level than SEVERE. For information about Oracle Fusion Middleware logging functionality, see the "Managing Log Files and Diagnostic Data" chapter of *Administering Oracle Fusion Middleware*.
2. Look for struck threads, as described in the "Monitor server performance" topic in the *Oracle WebLogic Server Administration Console Online Help*. If you find a stuck thread, examine the thread stack dump.
3. If you observe an HTTP 403 or 404 error on partial page rendering (PPR), then it is most like a bug.

Browser Displays an ADF_Faces-60098 Error

The application returns a runtime exception in a place that was not expected and is not handled.

Cause

ADF Faces has received an unhandled exception in some phase of the lifecycle and will abort the request handling.

Solution

To resolve this problem:

1. This is most likely a logic error in the application.
2. Verify that the server load or the application is not in distress.

Browser Displays an HTTP 404 or 500 Error

The application does not navigate to the expected page and displays an HTTP 404 file not found error or an HTTP 500 internal server error.

Cause

The cause may be traced to the application server.

Solution

To resolve this problem:

1. Verify that the application server is running and that the application is not in distress, as described in the "Monitor server performance" and "Servers: Configuration: Overload" topics in the *Oracle WebLogic Server Administration Console Online Help*.
2. Check for hung threads.

Browser Fails to Navigate Between Pages

The application fails to navigate to and open an expected target web page.

Cause

The cause may depend on the application design or the cause may be traced to the application server.

Solution

To resolve this problem:

1. Check for unhandled exceptions specific to an ADF Faces lifecycle thread, as described in [Browser Displays an ADF_Faces-60098 Error](#).
2. Look for HTTP 404 or 505 errors, as described in [Browser Displays an HTTP 404 or 500 Error](#).

Using My Oracle Support for Additional Troubleshooting Information

My Oracle Support is Oracle's official electronic on-line support service. You can log a Service Request to receive personalized, proactive, and collaborative support to resolve issues specific to Oracle ADF.

You can use My Oracle Support (formerly MetaLink) to help resolve Oracle Fusion Middleware problems. My Oracle Support contains several useful troubleshooting resources, such as:

- Knowledge base articles
- Community forums and discussions
- Patches and upgrades
- Certification information

Note:

You can also use My Oracle Support to log a service request.

You can access My Oracle Support at <https://support.oracle.com>.