Oracle® Fusion Middleware

Developing Web User Interfaces with Oracle ADF Faces

12*c* (12.1.2)

E23447-03

April 2014

Documentation for developers that describes how to create web-based applications using ADF Faces components and the supporting architecture.

ORACLE®

Oracle Fusion Middleware Developing Web User Interfaces with Oracle ADF Faces 12*c* (12.1.2)

E23447-03

Primary Authors: Robin Whitmore (lead), Peter Jew, Kathryn Munn, Walter Egan, Himanshu Marathe, Ralph Gordon, Cindy Hall, Michele Whittaker

Contributing Author: Poh Lee Tan

Contributors: ADF Faces development team, Frank Nimphius, Laura Akel, Katia Obradovic-Sarkic

Contents

Part II Understanding ADF Faces Architecture

4 Using ADF Faces Client-Side Architecture

5 Using the JSF Lifecycle with ADF Faces

6 Handling Events

7 Validating and Converting Input

8 Rerendering Partial Page Content

Part III Creating Your Layout

9 Organizing Content on Web Pages

Part IV Using Common ADF Faces Components

11 Using Input Components and Defining Forms

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

13 Using List-of-Values Components

14 Using Query Components

15 Using Popup Dialogs, Menus, and Windows

16 Using Menus, Toolbars, and Toolboxes

17 Using a Calendar Component

18 Using Output Components

19 Displaying Tips, Messages, and Help

20 Working with Navigation Components

21 Determining Components at Runtime

Part V Using ADF Data Visualization Components

22 Introduction to ADF Data Visualization Components

23 Using Graph Components

24 Using Gauge Components

25 Using Pivot Table Components

26 Using Gantt Chart Components

27 Using Timeline Components

28 Using Map Components

29 Using Hierarchy Viewer Components

30 Using Treemap and Sunburst Components

Part VI Completing Your View

31 Customizing the Appearance Using Styles and Skins

35 Allowing User Customization on JSF Pages

36 Adding Drag and Drop Functionality

37 Using Different Output Modes

38 Using the Active Data Service with an Asynchronous Backend

Part VII Appendixes

A ADF Faces Configuration

E Quick Start Layout Themes

F Code Samples

G Troubleshooting ADF Faces

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 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:

- *Creating ADF Skins with Oracle ADF Skin Editor*
- *Understanding Oracle Application Development Framework*
- *Developing Applications with Oracle JDeveloper*
- *Developing Fusion Web Applications with Oracle Application Development Framework*
- *Administering Oracle ADF Applications*
- *Developing Oracle ADF Mobile Browser 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 12*c* 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.
italic	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

The following topics introduce the new and changed features of ADF Faces and other significant changes, which are described in this guide. This document is the new edition of the document formerly titled *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

New and Changed Features for Release 12*c* (12.1.2)

Oracle Fusion Middleware Release 12*c* (12.1.2) of Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) includes the following new and changed development features, which are described in this guide.

- New Skyros skin. All images of components at runtime have been replaced to reflect the new Skyros skin. In addition, the following specific changes have been made:
  - Section 23.4.1.1, "How to Globally Set Graph Font Using a Skin" removed references to the `blafplus-rich.desktop` skin.
- Facelets support. All chapters have been revised for Facelets.
- New JDeveloper window names, context menu options, and other user interface changes in JDeveloper are reflected in all chapters of this guide.
- Button and link components are now consolidated into the `af:button` and `af:link` components. All code examples in the book have been updated to reflect these new components.
- New `target` tag allows you to explicitly state which target components you want executed and rendered when a specific event (or events) is fired by a component. This provides a more fine-grained ability to control partial page rendering. For more information, see Chapter 8, "Rerendering Partial Page Content."
- The `convertTimeZone` converter can interpret four-digit year values when using a two-digit year pattern, in conjunction with the `inputText` and `inputDate` components. See Section 7.3.4, "What You May Need to Know About Date Time Converters."
- New `panelGridLayout, gridRow`, and `gridCell` components. See Section 9.3, "Arranging Content in a Grid."
- New `panelDrawer` and `panelSpringboard` components. See Section 9.10, "Displaying or Hiding Contents in Panels."
- The new `placeholder` and `usage` attributes for input components:

- For the `inputText` component, see Section 11.3.1, "How to Add an inputText Component."
- For the `inputColor` component, see Section 11.5.1, "How to Add an inputColor Component."
- For the `inputDate` component, see Section 11.5.2, "How to Add an InputDate Component."

- Custom list of time zones can be configured for the `inputDate` component. See Section 11.5.4, "What You May Need to Know About Creating a Custom Time Zone List."
- New `animationInterval` attribute on the `inputNumberSlider` and `inputRangeSlider` components. See Section 11.4.1, "How to Add an inputNumberSlider or an inputRangeSlider Component."
- Support for an indeterminate (or mixed) state on the `selectBooleanCheckbox` component. See Section 11.6, "Using Selection Components."
- Multiple file upload support on the `inputFile` component. See Section 11.9, "Using File Upload."
- New `codeEditor` component that renders a widget for code editing. See Section 11.10, "Using Code Editor."
- New `allDayActivityOrder`, `hourZoom`, and `timeSlotsPerHour` attributes for the `calendar` component. See Section 17.3.1, "How to Configure the Calendar Component."
- The `table` and `treeTable` components can now navigate through "pages" of rows, rather than continuous scrolling. See Section 12.2.2, "Content Delivery."
- The `table` and `treeTable` components now allow freezing of columns starting from the end column. See Section 12.3.2, "Formatting Tables."
- The `table` and `treeTable` components can now have individual rows span multiple columns. See Section 12.3.3, "Formatting Columns."
- New `listView` and `listItem` components provide a simplified, one-column table. See Section 12.10, "Displaying a Collection in a List."
- New `selected` attribute for `button` and `link` components that, if set to `true`, styles the component as selected it renders. The `iconPosition` attribute for these components also supports additional values (`top` and `bottom`) to determine where the component renders an icon if one is specified. See Section 20.3, "Using Buttons and Links for Navigation."
- Changes to `navigationPane` component:
  - Ability to determine the size of the tabs rendered by a `navigationPane` component. See Section 20.8.4, "What You May Need to Know About the Size of Navigation Tabs."
  - Display overflow of tabs like a conveyor belt. See Section 20.8.5, "What You May Need to Know About Navigation Tabs in a Compressed Layout."
- New `dynamicComponent` allows the exact components to be rendered to be determined at runtime. See Chapter 21, "Determining Components at Runtime."
- New `timeline` component that 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. See Chapter 27, "Using Timeline Components." Chapter 22, "Introduction to ADF Data Visualization Components"

was also revised to add an introduction to this new component.

- New `treemap` and `sunburst` components that display quantitative hierarchical data across two dimensions, represented visually by size and color. See the following:
  - Chapter 22, "Introduction to ADF Data Visualization Components"
  - Chapter 30, "Using Treemap and Sunburst Components."
  - Section 33.3.3, "ADF Data Visualization Components Accessibility Guidelines"
  - Section 36.8.3, "Adding Drag and Drop Functionality for DVT Hierarchy Viewers, Sunbursts, and Treemaps"
- HTML5 support. If the client's browser supports it, HTML5 is now the default image format, instead of Flash, for the following DVT components:
  - `gauge`. See Section 24.2.5, "What You May Need to Know About Gauge Image Formats."
  - `graph`. See Section 23.2.6, "What You May Need to Know About Graph Image Formats."
  - `thematicMap`. See Section 28.6.6, "What You May Need to Know About Thematic Map Image Formats."

  HTML5 is also the default image format for all new DVT components, including sunbursts, timelines, and treemaps.
- New `positionHint` and `maxWidth` attributes on the `graph` component. See Section 23.3.4, "How to Customize Graph Legends."
- Data marker selection support on graphs. See Section 23.7.8, "Adding Data Marker Selection Support for Graphs."
- New features on the `pivotTable` component:
  - Single-click editing. See Section 25.1.2.3, "Editing Data Cells."
  - Context menu options for column and header layer sorting. See Section 25.1.2.4, "Data and Header Sorting."
  - New page control alternative to scrollbars, and attributes available to set the start position for row and column data cells or headers. See Section 25.1.2.6, "Scrolling and Page Controls" and Section 25.2.6, "What You Many Need to Know About Displaying Large Data Sets."
  - Persistent header layer labels when displaying large data sets. See Section 25.1.2.7, "Persistent Header Layers."
  - Split view configuration of the data in large data sets. See Section 25.1.2.8, "Split View of Large Data Sets."
  - Header cell word wrapping with long header titles. See Section 25.1.2.10, "Header Cell Word Wrapping."
  - Active Data Service (ADS) support. See Section 25.1.2.11, "Active Data Support (ADS)."
  - Declarative styling attributes for header and data cells. See Section 25.2.4, "Configuring Pivot Table Display Size and Style."
  - Mobile device support. See Section 25.2.7, "What You May Need to Know About Pivot Tables on Touch Devices."
  - Pivot filter bar. See Section 25.4.2, "Using a Pivot Filter Bar with a Graph."

- Support for printable pages. See Section 25.5.4, "Displaying Pivot Tables in Printable Pages."

- Drag and drop support for `hierarchyViewer` component. See Section 36.8.3, "Adding Drag and Drop Functionality for DVT Hierarchy Viewers, Sunbursts, and Treemaps."
- The `popup` component is now a valid root for the `showPrintablePageBehavior` tag. See Section 37.2, "Displaying a Page for Print."
- Only postback values of editable components when those values have changed since the last request. See Section A.2.3.16, "Postback Payload Size Optimization."
- Display or hide ADF version information on an HTML page. See Section A.2.3.19, "Version Number Information."
- Support for Oracle Traffic Director by the ADF Faces Caching Filter (ACF). See Section A.2.3.21, "ADF Faces Caching Filter."
- Invalid browser notification. See Section A.2.3.23, "Internet Explorer Compatibility View Mode."
- Session timeout when a warning dialog is displayed. See Section A.2.3.24, "Session Timeout Warning."
- Removal of dynamic URL parameters in URLs. See Section A.2.3.26, "Clean URLs."
- Ability to disable the splash screen displayed when loading a page. See Section A.2.3.27, "Page Loading Splash Screen."
- Globally control over how certain layout components and tables handle being stretched. See Section A.2.3.29, "Geometry Management for Layout and Table Components."
- Content delivery network (CDN) support. See Appendix A.4.4, "Using Content Delivery Networks."
- Increased support for tablet and mobile devices. See Appendix D, "Creating Web Applications for Touch Devices Using ADF Faces."

Other Significant Changes in this Document for Release 12*c* (12.1.2)

For Release 12*c* (12.1.2), this document has been updated in several ways, to include corrections and clarifications. In addition, the following significant changes or additions have occurred in these sections:

- Chapters now include a "Use Cases and Examples" section.
- All Introduction sections in chapters have been renamed to "About..."
- Revised Section 7.4, "Creating Custom ADF Faces Converters" for clarity.
- Updated code examples in Section 7.4.1, "How to Create a Custom ADF Faces Converter."
- Removed obsolete text in Section 7.5.1, "How to Add Validation."
- Moved the "Displaying Images in a Carousel" section from the "Using Output Components" chapter to "Using Tables, Trees, and Other Collection-Based Components." See Section 12.11, "Displaying Images in a Carousel."
- Revised Chapter 19, "Displaying Tips, Messages, and Help" for clarity and to remove obsolete content.

- Revised Chapter 32, "Internationalizing and Localizing Pages" to describe the configuration you make in an `adf-config.xml` file if you want to override a resource bundle in a customizable application. See Section 32.3.8, "What You May Need to Know About Overriding a Resource Bundle in a Customizable Application."
- Revised Chapter 22, "Introduction to ADF Data Visualization Components" to remove obsolete content regarding embedded fonts for the `graph` and `gauge` components. In addition, the DVT components that support the Active Data Service (ADS) are listed.
- Revised various sections in Chapter 23, "Using Graph Components" to:
  - add more use cases.
  - correct errors in code examples and procedures.
  - reflect pre-existing components contained in the graph component.
  - add procedure for creating a time selector. See Section 23.7.3.3, "How to Add a Time Selector to a Graph."
- Revised various sections in Chapter 24, "Using Gauge Components" to:
  - correct errors in code examples and procedures.
  - reflect pre-existing components contained in the gauge component.
  - move content from Section 24.4.3, "How to Format Gauge Text" ti Section 24.4.4, "How to Specify a Gauge Text Resource."
  - describe how to use skinning keys for various functionality.
- Removed the following sections from Chapter 24, "Using Gauge Components"
  - "How to Specify the Layout of Gauges in a Gauge Set"
  - "What You May Need to Know About Configuring Gauge Set Display"
- Revised Section 33.3.3, "ADF Data Visualization Components Accessibility Guidelines" to document guidelines for the `treemap` and `sunburst` components.
- Revised various sections in Chapter 29, "Using Hierarchy Viewer Components" to:
  - correct errors in code examples and procedures
  - describe how to use skinning keys for various functionality.
- Removed obsolete section in Chapter 29, "Using Hierarchy Viewer Components": "How to Configure a Hierarchy Viewer to Invoke a Context Menu."
- Consolidated drag and drop features for graph and gantt components into Section 36.8, "Adding Drag and Drop Functionality for DVT Components."
- Revised Section 37.3, "Creating Emailable Pages" to add to the list of supported email clients and to add information about the `table` and `treeTable` components.
- Revised Section A.2.3.11, "Test Automation" to document coding errors that will produce assertion failed errors only after test automation is enabled.
- Added new appendix to document troubleshooting the application user interface. See Appendix G, "Troubleshooting ADF Faces."

Part I

Getting Started with ADF Faces

Part I contains the following chapters:

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:

- Section 1.1, "About ADF Faces"
- Section 1.2, "ADF Faces Framework"
- Section 1.3, "ADF Faces Components"

1.1 About ADF Faces

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 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 more about JSF, see `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`.

1.2 ADF Faces Framework

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

- Built to the JSF 2.0 specification

  ADF Faces supports JSF 2.0, including Facelets. Several of the new JavaServer Faces 2.0 features have parallel functionality in ADF Faces. To understand the new 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. For more information, see Section 1.3, "ADF Faces Components." For more information about running ADF Faces on tablets, see Appendix D, "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). For more information, see Chapter 8, "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. For more information, see Chapter 4, "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. For more information, see Chapter 5, "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 more information about events, see Chapter 6, "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. For more information, see Section 8.5, "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 in written JavaScript and validation errors caught on the client-side can be processed without a round-trip to the server. For more information, see Chapter 7, "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. For more information, see Chapter 38, "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. For more information, see Section 9.2.1, "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. For more information, see Chapter 10, "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. For more information, see Part V, "Using ADF Data Visualization Components".

- Skinning

  You can create your own look and feel by implementing skins for ADF Faces components. Oracle provides a stand-alone skin editor, where you can declaratively create and modify your skins. For more information, see Chapter 31, "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. For more information, see Chapter 37, "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. For more information, see Chapter 32, "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. For more information, see Chapter 33, "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. For more information, see Chapter 35, "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. For more information, see Chapter 36, "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 more information about using ADF Faces with the ADF technology stack, see *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 more information about using JDeveloper, see Chapter 3, "Getting Started with ADF Faces and JDeveloper."

1.3 ADF Faces Components

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 more information about layout components and geometry management, see Chapter 9, "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. For more information, see Chapter 16, "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. For more information, see Chapter 15, "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. For more information, see Chapter 18, "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 more information about input components, see Chapter 11, "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. For more information, see Chapter 13, "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. For more information, see Chapter 12, "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. For more information, see Chapter 22, "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. For more information, see Chapter 14, "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. For more information, see Chapter 17, "Using a Calendar Component." The carousel component allows you to display a collection of images in a scrollable manner. For more information, see Section 12.11, "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. For more information, see Chapter 19, "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. For more information, see Chapter 20, "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. For more information, see Chapter 20, "Working with Navigation Components."
  - Images and icon components: Allow you to display images as simple as icons, to as complex as video. For more information, see Chapter 18, "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 Demo Application

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

This chapter contains the following sections:

- Section 2.1, "About the ADF Faces Demonstration Application"
- Section 2.2, "Downloading and Installing the ADF Faces Demo Application"

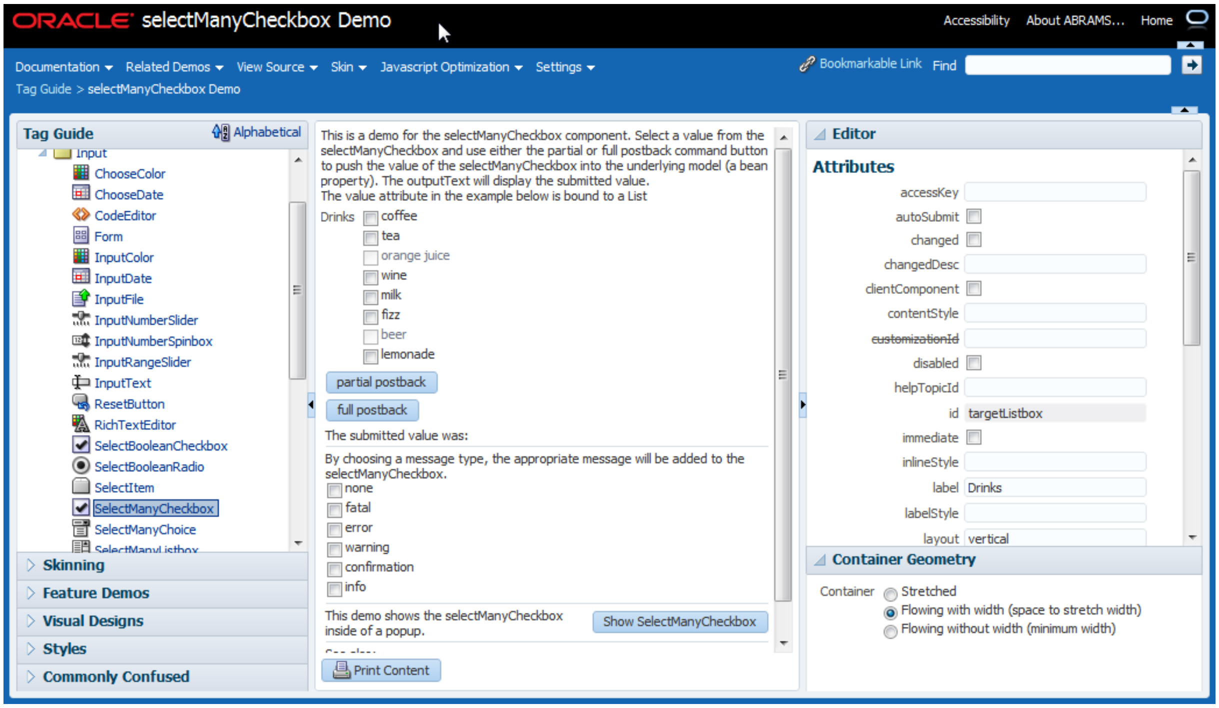
2.1 About the ADF Faces Demonstration Application

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

The demo application contains the following:

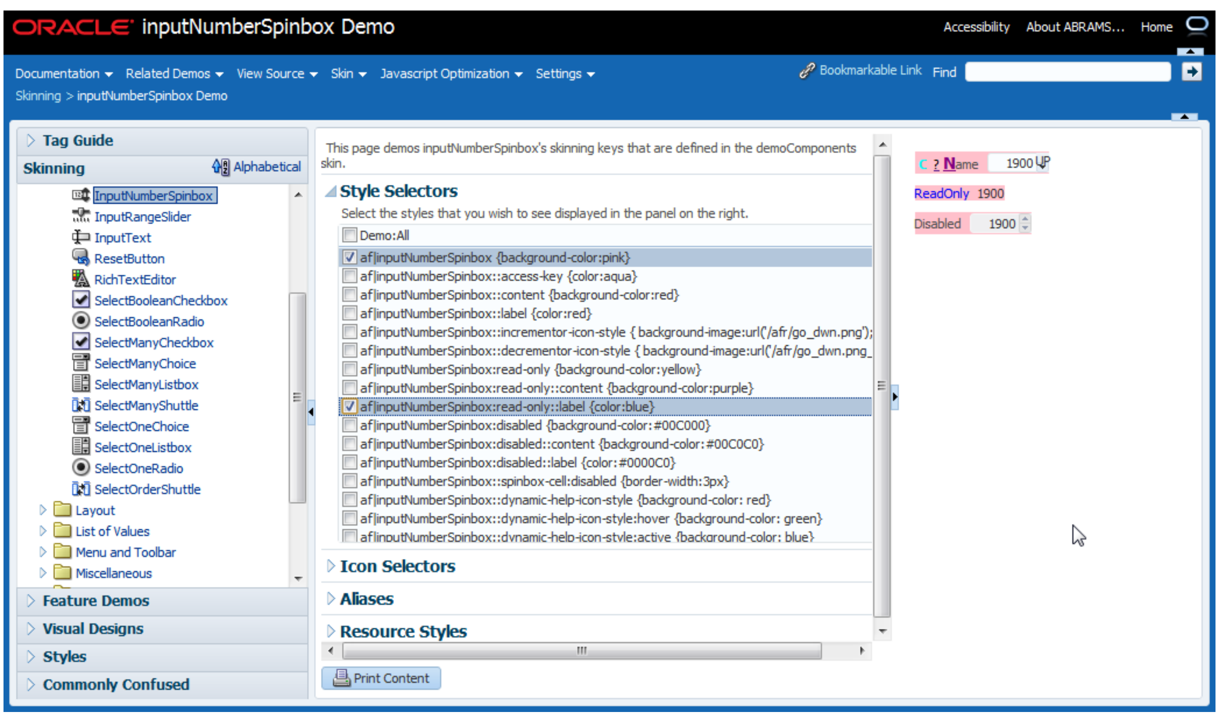
- Tag guide: Demonstrations of ADF Faces components, validators, converters, and miscellaneous tags, along with a property editor to see how changing attribute values affects the component. Figure 2–1 shows the demonstration of the `selectManyCheckbox` component. Each demo provides a link to the associated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

Figure 2–1 Tag Demonstration



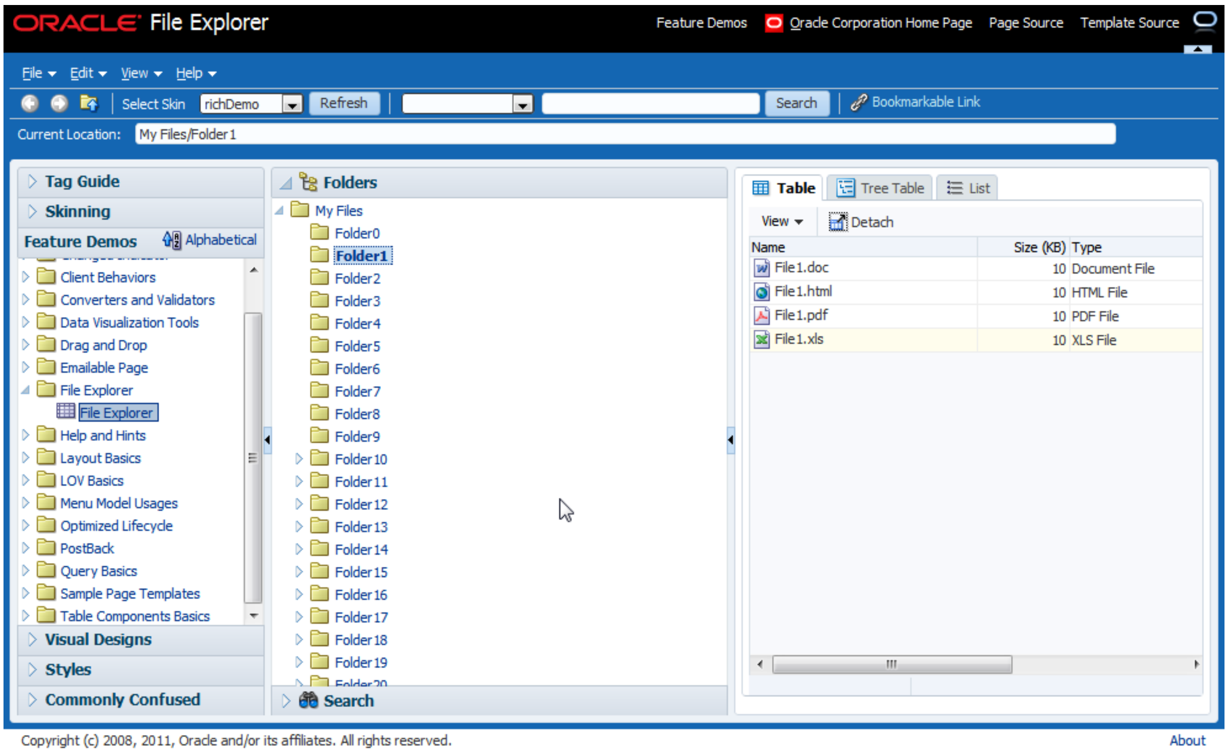
- Skinning: Demonstrations of skinning on the various components. You can see, for example, how changing style selectors affects how a component is displayed. Figure 2–2 shows how setting certain style selectors affects the `inputNumberSpinbox` component.

Figure 2–2 Skinning Demonstration



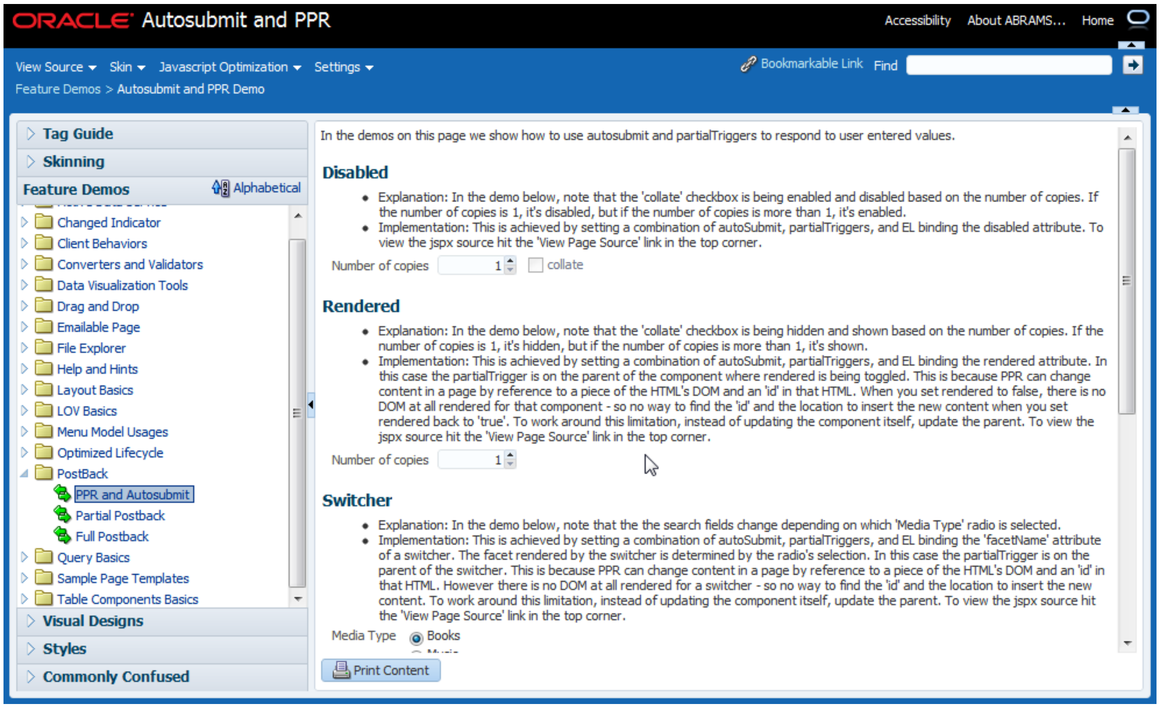
- Feature demos: Various pages that demonstrate different ways you can use ADF components. For example, the File Explorer is an application with a live data model that displays a directory structure and allows you to create, save, and move directories and files. This application is meant to showcase the components and features of ADF Faces in a working application, as shown in Figure 2–3.

Figure 2–3 File Explorer Application



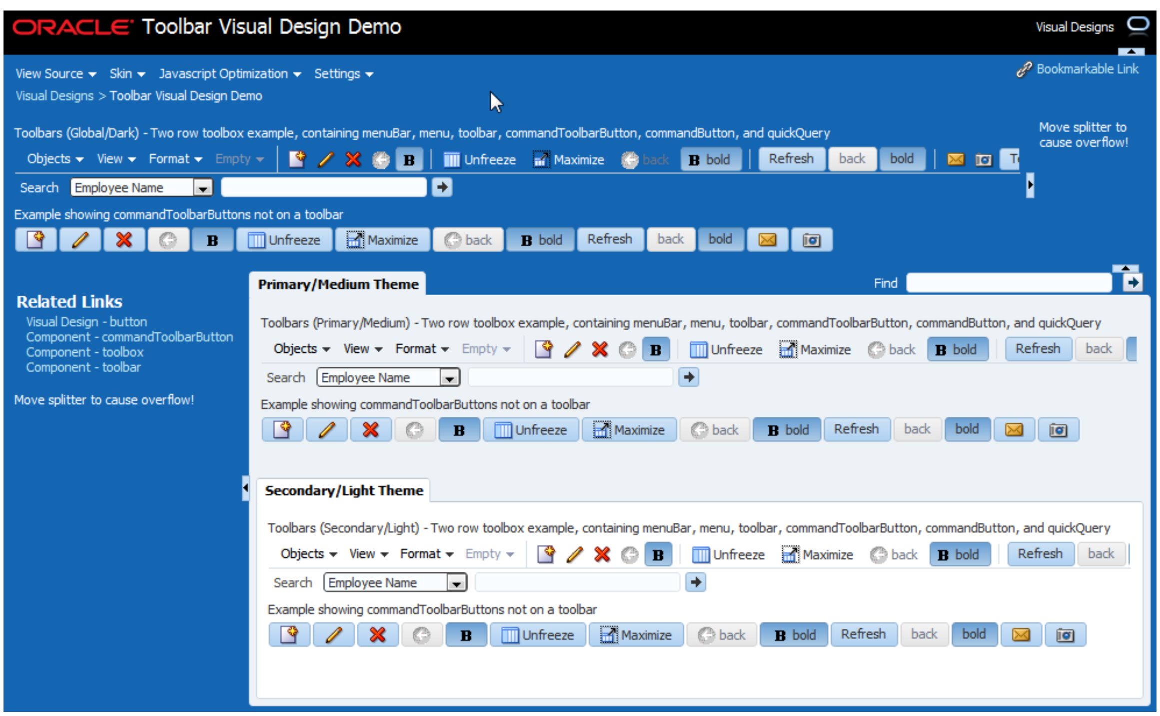
Other pages demonstrate the main architectural features of ADF Faces, such as layout components, Ajax postback functionality, and drag and drop. Figure 2–4 shows the demonstration on using the `AutoSubmit` attribute and partial page rendering.

Figure 2–4 Framework Demonstration



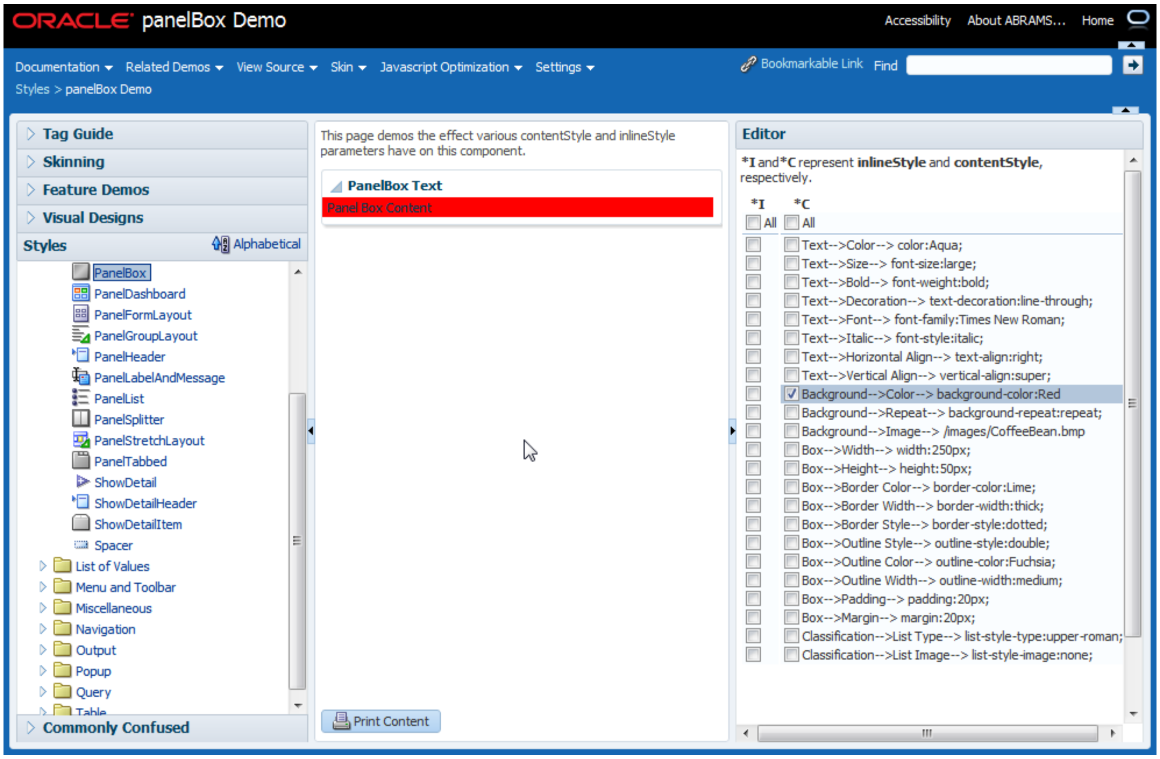
- Visual designs: Demonstrations of how you can use types of components in different ways to achieve different UI designs. Figure 2–5 shows how you can achieve different looks for a toolbar.

Figure 2–5 Toolbar Design Demonstration



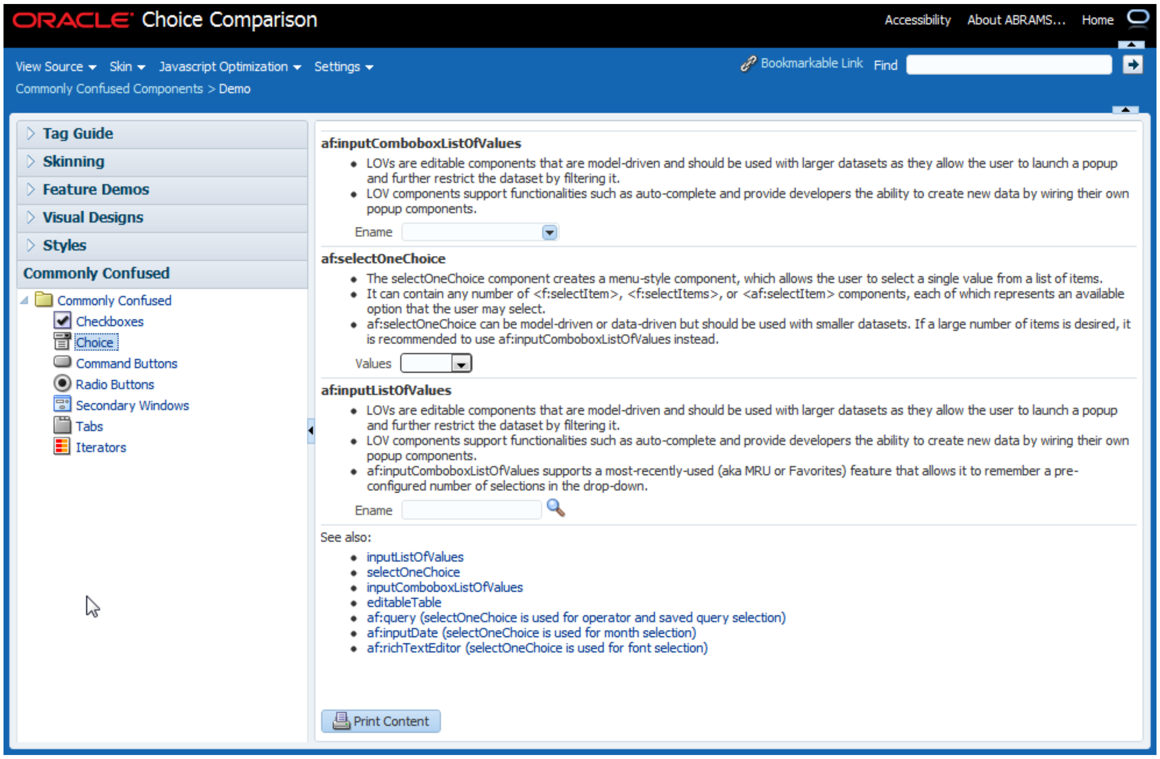
- Styles: Demonstration of how setting inline styles and content styles affects components. Figure 2–6 shows different styles applied to the `panelBox` component.

Figure 2–6 Styles Demonstration



- Commonly confused components: A comparison of components that provide similar functionality. Figure 2–7 shows the differences between the various components that display selection lists.

Figure 2–7 Commonly Confused Components

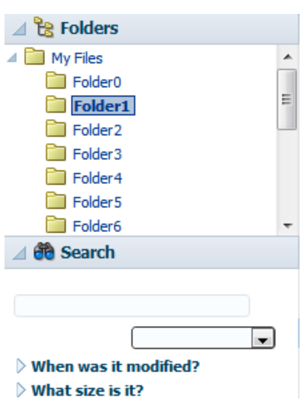


Because the File Explorer is a complete working application, many sections in this guide use that application to illustrate key points, or to provide code samples. The source for the File Explorer application can be found in the `fileExplorer` directory.

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 more information about layout components, see Chapter 9, "Organizing Content on Web Pages." For more information about using templates, see Chapter 10, "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 2–8.

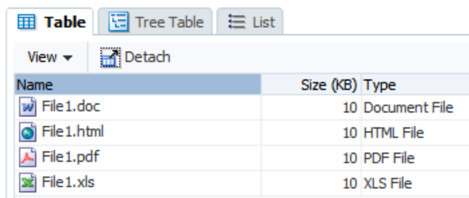
Figure 2–8 Directory Structure Panel and Search Panel



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 more information about using `panelAccordion` components, see Section 9.10, "Displaying or Hiding Contents in Panels." For more information about using input components, see Chapter 11, "Using Input Components and Defining Forms." For more information about using buttons, see Chapter 20, "Working with Navigation Components." For more information about using tables and trees, see Chapter 12, "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 2–9.

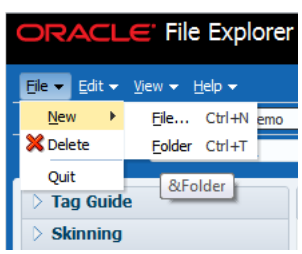
Figure 2–9 Directory Contents in Tabbed Panels



The table and tree table have built-in toolbars that allow you to manipulate how the contents are displayed. In the table an 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 more information about using tabbed panes, see Section 9.10, "Displaying or Hiding Contents in Panels." For more information about table and tree table toolbars, see Section 12.9, "Displaying Table Menus, Toolbars, and Status Bars." For more information about enabling drag and drop, see Chapter 36, "Adding Drag and Drop Functionality." For more information about using context menus and popup windows, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

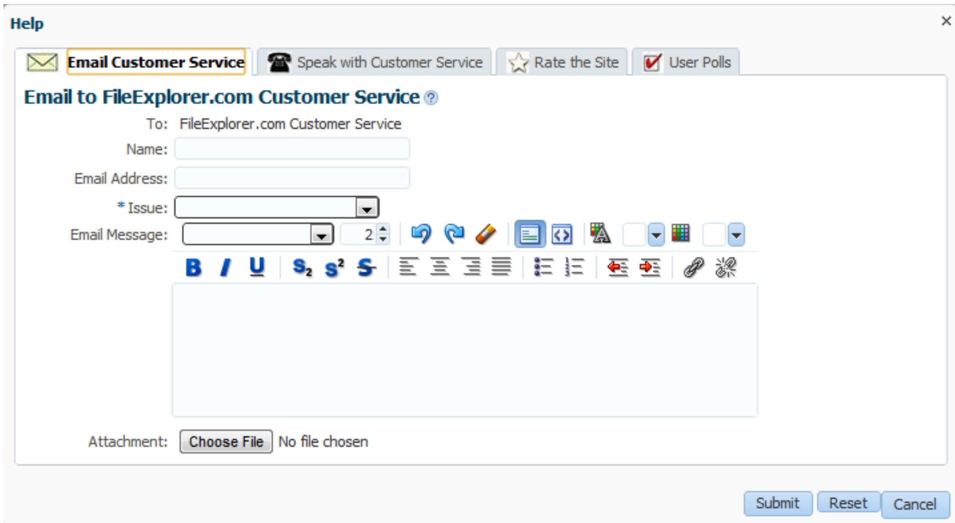
The top of the File Explorer application contains a menu and a toolbar, as shown in Figure 2–10.

Figure 2–10 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 2–11.

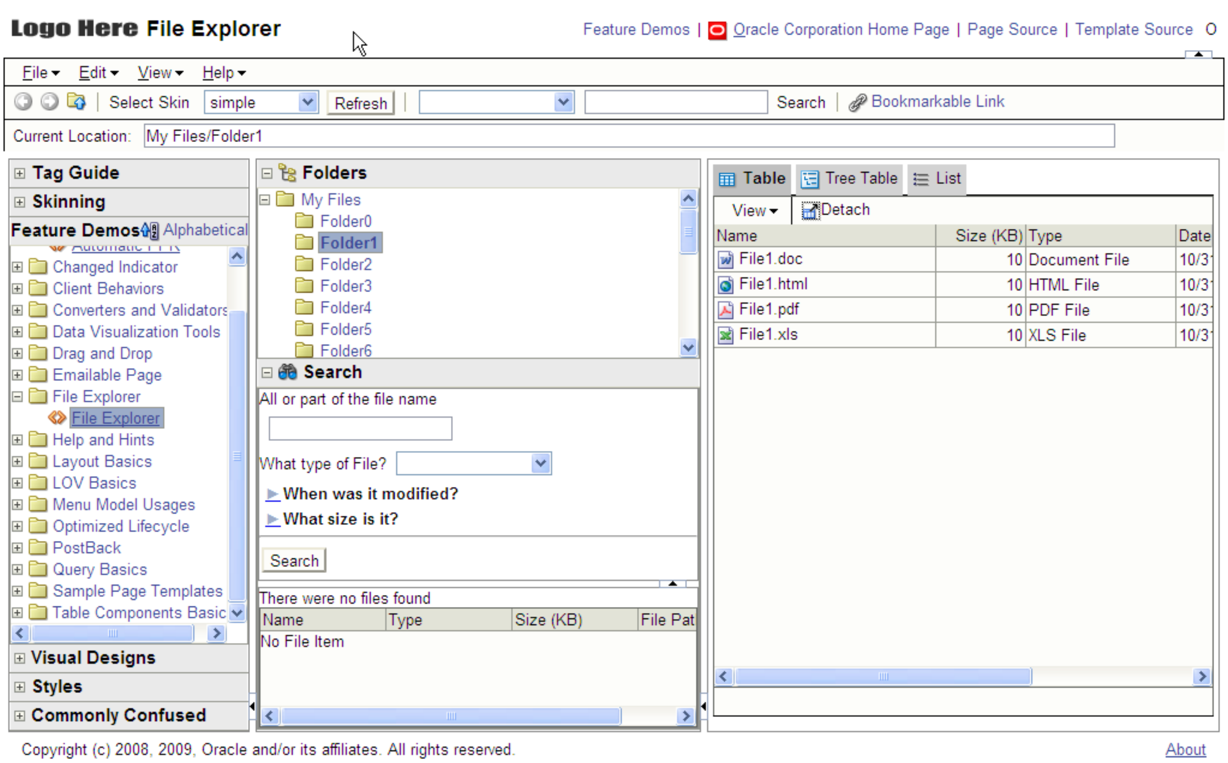
Figure 2–11 Help System



The help system consists of a number of forms created with various input components, including a rich text editor. For more information about menus, see Section 16.2, "Using Menus in a Menu Bar." For more information about creating help systems, see Section 19.5, "Displaying Help for Components." For more information about input components, see Chapter 11, "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 2–12 shows the File Explorer application using the simple skin.

Figure 2–12 File Explorer Application with the Simple Skin



For more information about toolbars, see Section 16.3, "Using Toolbars." For more information about using skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

2.2 Downloading and Installing the ADF Faces Demo Application

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 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 Rich Client 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:

3.1 About Developing Declaratively in JDeveloper

Using JDeveloper 11*g* 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
- Designing page flows
- Designing and creating the pages using either Facelets or JavaServer Pages (JSPs).
- Deploying the application. For more information about deployment, see *Administering Oracle ADF Applications*. If your application uses ADF Faces with the ADF Model layer, ADF Controller, and ADF Business Components, see the "Deploying Fusion Web Applications" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

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

- Creating managed beans
- Creating and using EL expressions

- Viewing ADF Faces source code and Javadoc

JDeveloper also includes debugging and testing capabilities. For more information, see the "Testing and Debugging ADF Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

3.2 Creating an Application Workspace

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 *Developing Fusion Web Applications with Oracle Application Development Framework*.

3.2.1 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 Applications window, click **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:

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

3.2.2 What Happens When You Create an Application Workspace

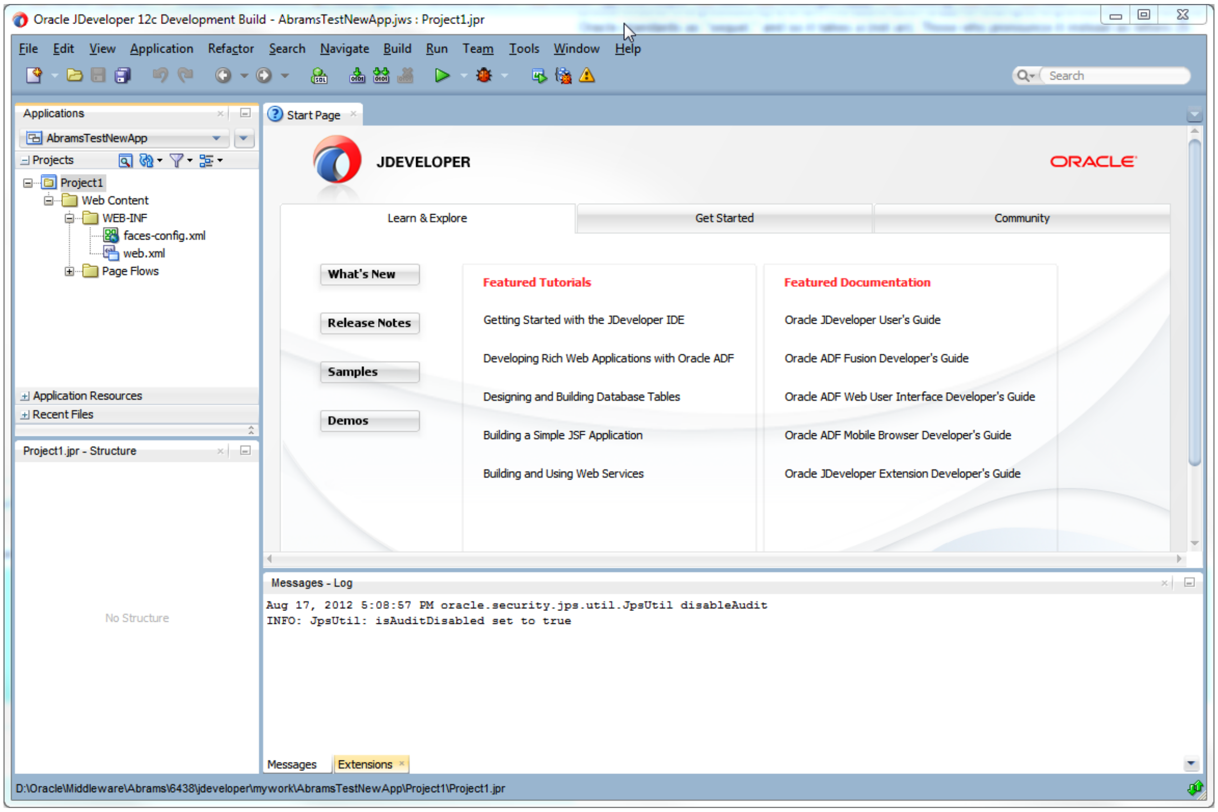
When you create an application workspace using the Custom template, and then select ADF Faces for your project, JDeveloper creates a project that contains all the source and configuration files needed for an ADF Faces application. Additionally, JDeveloper adds the following libraries to your project:

- JSF 2.1
- 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). Example 3–1 shows the `web.xml` file generated by JDeveloper when you create a new ADF Faces application.

Example 3–1 Generated web.xml File

```
<?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>
```

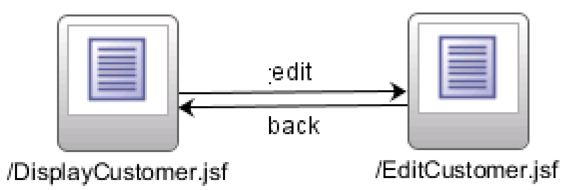
Configurations required for specific ADF Faces features are covered in the respective chapters of this guide. For example, any configuration needed in order to use the Change Persistence framework is covered in Chapter 35, "Allowing User Customization on JSF Pages." For comprehensive information about configuring an ADF Faces application, see Appendix A, "ADF Faces Configuration."

3.3 Defining Page Flows

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. For more information, see the "Getting Started with ADF Task Flows" chapter of *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 more information on how navigation works in a JSF application, see the Java EE 6 tutorial (`http://download.oracle.com/javaee/index.html`).

3.3.1 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:

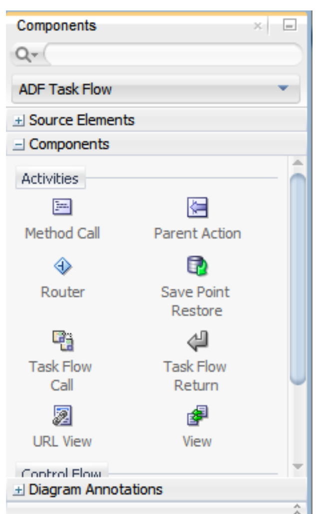
It may be helpful to have an understanding of page flows. For more information, see Section 3.3, "Defining Page Flows."

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 more information about using navigation components on a page, see Chapter 20, "Working with Navigation Components."

3.3.2 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. Example 3–2 shows the XML generated for the navigation rules displayed in Figure 3–2.

Example 3–2 Navigation Rules in faces-config.xml

```
<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>
```

3.4 Creating a View Page

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.

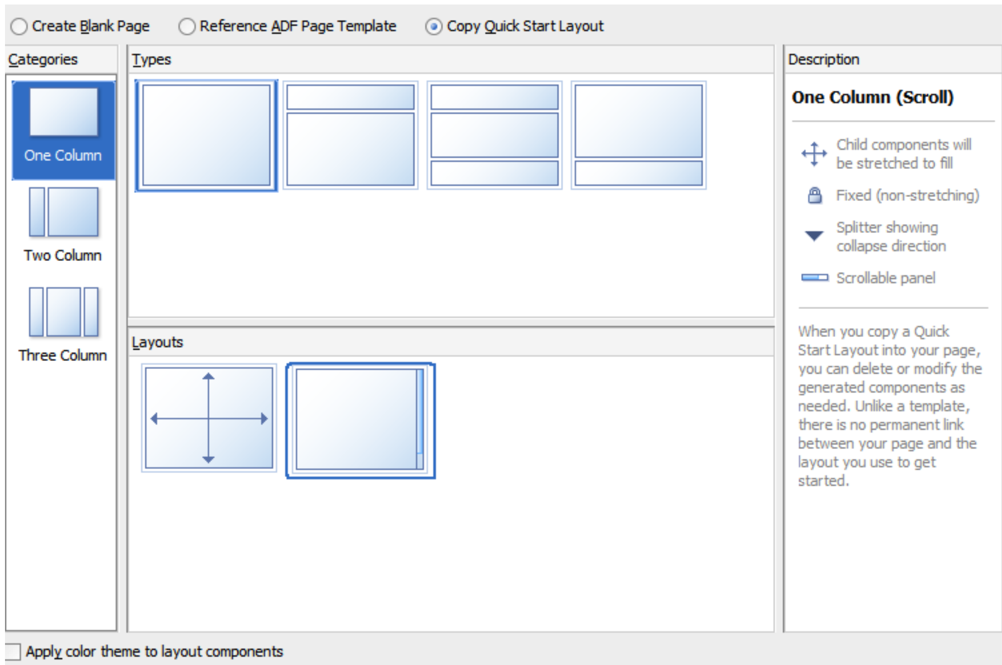
If instead you create a JSP page for an ADF Faces application, you create an XML-based JSP document, which uses the extension `*.jspx`. Using an XML-based document has the following advantages:

- 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 more information about the layout components, see Chapter 9, "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 Appendix E, "Quick Start Layout Themes." For more information about themes, see Chapter 31, "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 more information about creating and using templates, see Section 10.3, "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 more information about using backing beans with JSF pages, see Section 3.4.4, "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.

3.4.1 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. For more information, see Section 3.4, "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 more information about the Managed Bean page, which can be used to automatically create a backing bean and associated bindings, see Section 3.4.4, "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. For more information, see Chapter 35, "Allowing User Customization on JSF Pages."

3.4.2 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.

Example 3–3 shows a Facelets page when it is first created by JDeveloper.

Example 3–3 Declarative Facelets Page Source 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="d1">
        <af:form id="f1"></af:form>
    </af:document>
</f:view>
```

Example 3–4 shows a `.jspx` page when it is first created by JDeveloper.

Example 3–4 Declarative JSPX Page Source 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. Example 3–5 shows the generated code 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.

Example 3–5 Two-Column Layout

```
<?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" halign="stretch"
                                     valign="stretch" id="gc1">
                            <!-- Left -->
                        </af:gridCell>
                        <af:gridCell width="100%" halign="stretch"
                                     valign="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. Example 3–6 shows the code created for a backing bean for a page.

Example 3–6 Declarative Backing Bean Source Created by JDeveloper

```
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 d1;

    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 JEWT

JDeveloper also adds entries to the `web.xml` file. Example 3–7 shows the `web.xml` file created once you create a JSF page.

Example 3–7 Code in the web.xml File After a JSF Page is Created

```
<?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>
  <servlet>
    <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 in Example 3–8.

Example 3–8 Generated faces-config.xml File

```
<?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 Example 3–9.

Example 3–9 Generated trinidad-config.xml File

```
<?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.

3.4.3 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 code shown in Example 3–10 into your `web.xml` page.

Example 3–10 web.xml Code for Running Both JSP and Facelets Pages Using the Facelets Engine

```
<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 in Example 3–11.

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.

Example 3–11 Add this web.xml Code When Using Facelets Engine

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

> **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.

3.4.4 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 more information about managed beans, see Section 3.6, "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:

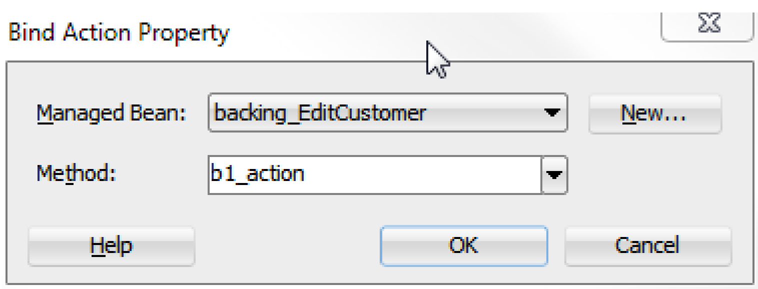
- Creates a JavaBean using the same name as the JSF file, and places it in the `view.backing` package (if you elect to have JDeveloper create a backing bean).
- Creates a managed bean entry in the `faces-config.xml` file for the backing bean. By default, the managed bean name is `backing_<page_name>` and the bean uses the `request` scope (for more information about scopes, see Section 5.6, "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. For more information, see Section A.3.1, "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.

Example 3–12 shows the code on a JSF page that uses automatic component binding, and contains form, inputText, and button components.

Example 3–12 JSF Page Code with Automatic Component Binding

```
<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>
```

Example 3–13 shows the corresponding code on the backing bean.

Example 3–13 Backing Bean Code Using Automatic Component Binding

```
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;
    }
}
```

Example 3–14 shows the code added to the `faces-config.xml` file to register the page's backing bean as a managed bean.

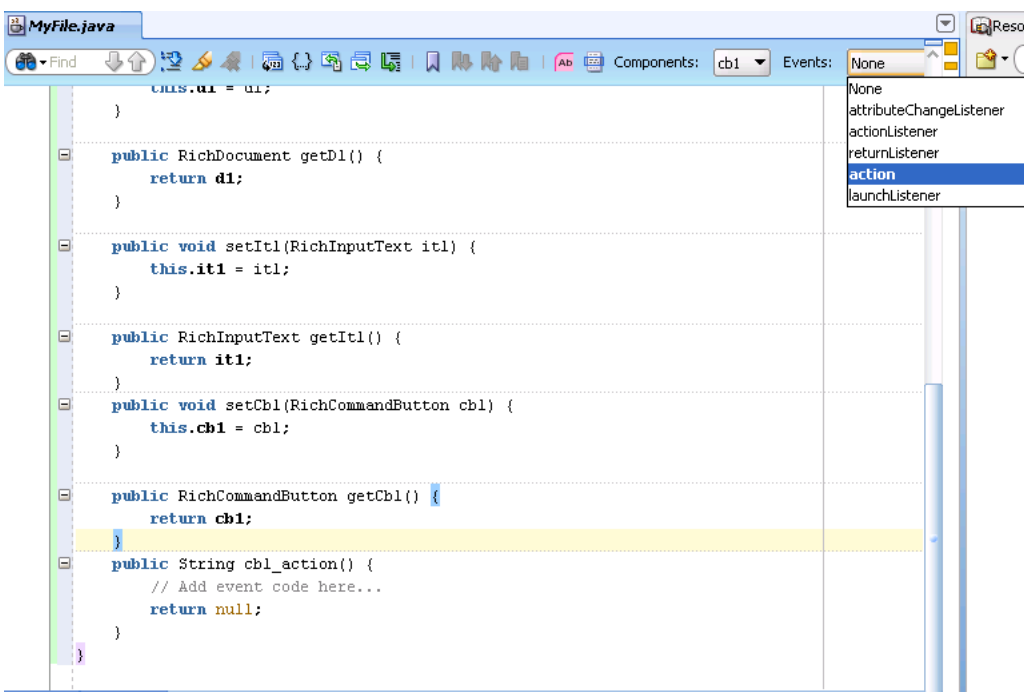
Example 3–14 Registration for a Backing Bean

```
<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 more 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 more 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.

3.4.5 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 Part IV, "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. For more information, see Section 3.4, "Creating a View Page."

To add ADF Faces components to a page:

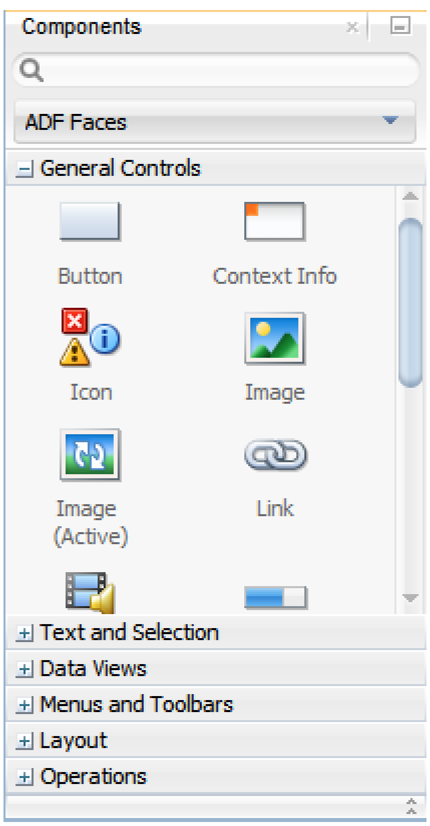
1. In the Applications window, double-click a JSF page to open it.
2. If the Components window is not displayed, from the menu choose **Window > Components**. By default, the Components window is displayed in the upper right-hand corner of JDeveloper.
3. In the Components window, use the dropdown menu 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:
>
> 1. Right-click the project node and choose **Project Properties**.
> 2. Select **JSP Tag Libraries** to add the ADF Faces library to the project. For help, click **Help** or press F1.
>
> For a JSPX file:
>
> 1. Right-click inside the Components window and choose **Edit Tab Libraries**.
> 2. 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.

3.4.6 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. Example 3–15 shows the code when you drop an Input Text and a Button component from the palette.

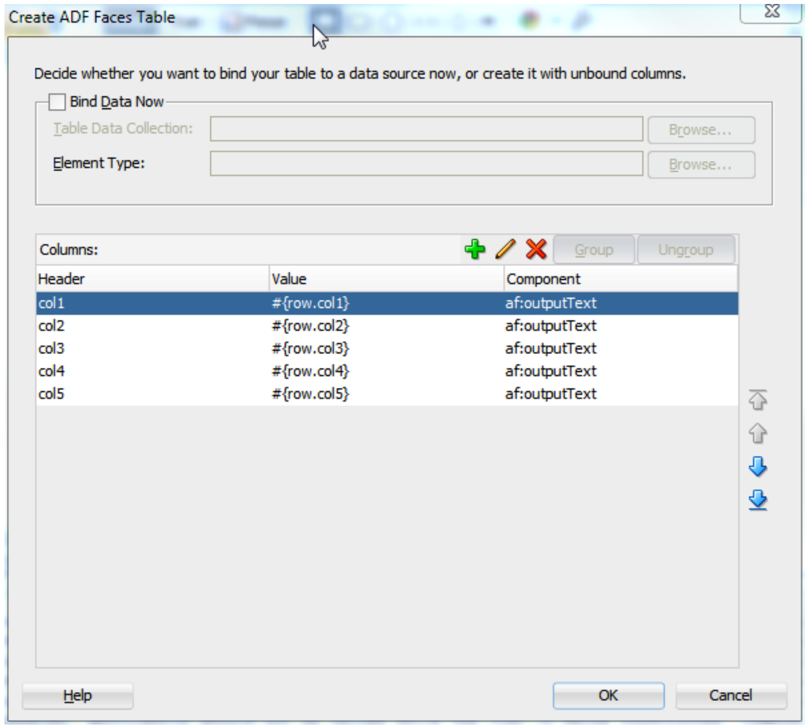
Example 3–15 JDeveloper Declaratively Adds Tags to a JSF Page

```
<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. For more information, see Section 3.4.4, "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



Example 3–16 shows 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.

Example 3–16 Declarative Code for a Table Component

```
<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>
```

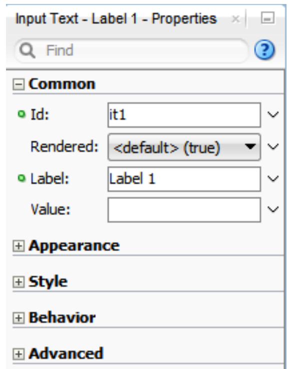
3.4.7 How to Set Component Attributes

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.

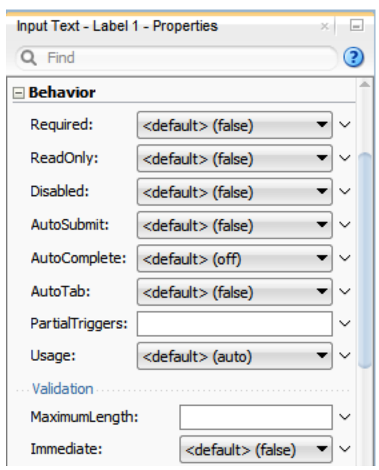
Figure 3–10 shows the Properties window displaying the attributes for an `inputText` component.

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 more information, see Section 3.4, "Creating a View Page."

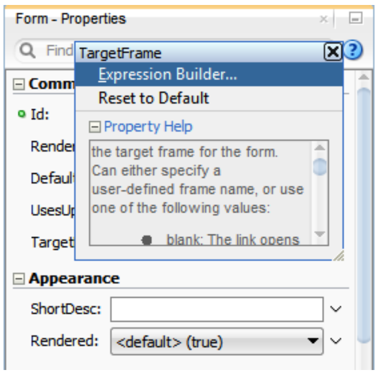
To set component attributes:

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 more information about using the Expression Builder, see Section 3.5, "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



3.4.8 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.

3.5 Creating EL Expressions

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 more information about creating and using managed beans, see Section 3.6, "Creating and Using Managed Beans." For more 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 it's 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 Section 3.5.2, "How to Use the EL Format Tags." For information about the time zone tags, see Section 11.5.3, "What You May Need to Know About Selecting Time Zones Without the inputDate Component."

3.5.1 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. For more information, see Section 3.5, "Creating EL Expressions."

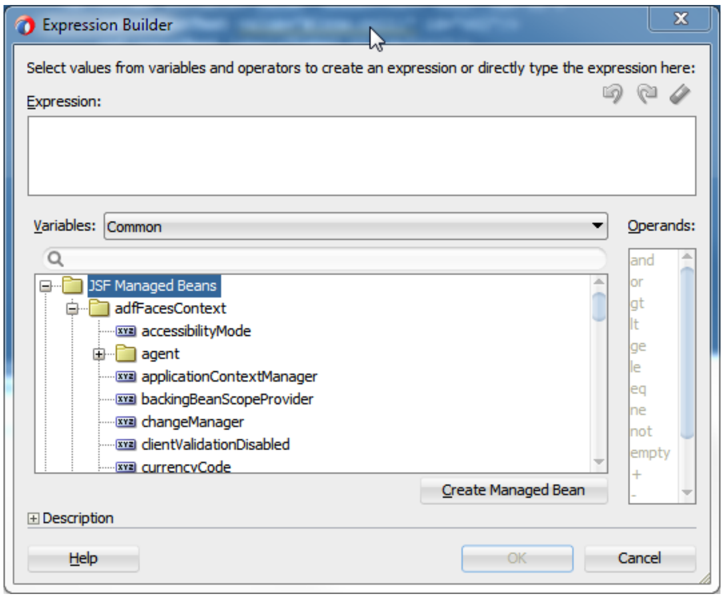
To use the Expression Builder:

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



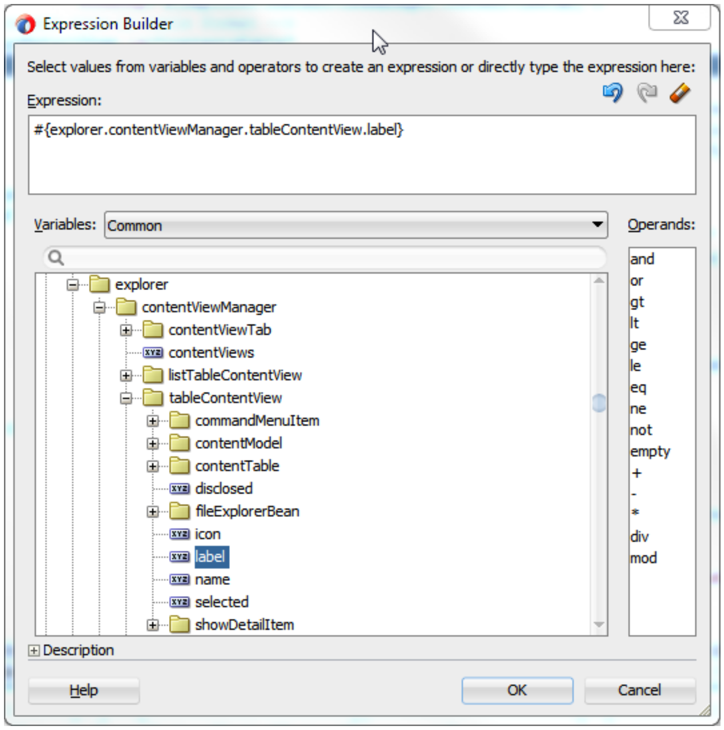
Tip: For more information about these objects, see the *Java API Reference for Oracle ADF Faces*.

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`.

3.5.2 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 in Example 3–17.

Example 3–17 Using the formatString Tag to Display a Message with a Parameter

```
<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. Example 3–18 shows a message that uses the `formatNamed2` tag to display the number of files on a specific disk. This message contains two parameters.

Example 3–18 Using the formatNamed2 Tag to Display a Message

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

3.5.3 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.

Example 3–19 shows how you can get a reference to an EL expression and return (or create) the matching object.

Example 3–19 Resolving an EL Expression from a Managed Bean

```
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);
 }
```

Example 3–20 shows how you can resolve a method expression.

Example 3–20 Resolving a Method Expression from a Managed Bean

```
public static Object resloveMethodExpression(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);
  }
```

Example 3–21 shows how you can set a new object on a managed bean.

Example 3–21 Setting 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);
  }
}
```

3.6 Creating and Using Managed Beans

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. For more information, refer to the "Using a Managed Bean in a Fusion Web Application" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

3.6.1 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

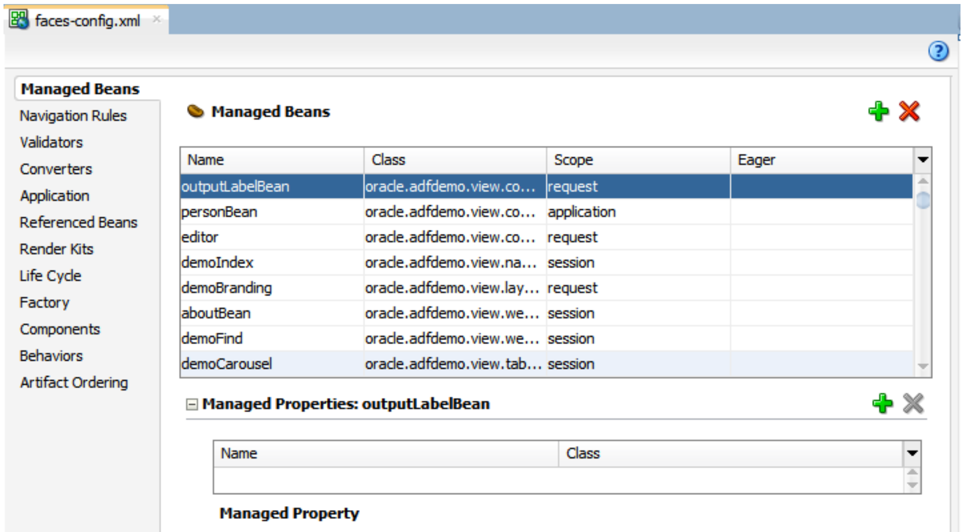
It may be helpful to have an understanding of managed beans. For more information, see Section 3.6, "Creating and Using Managed Beans."

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 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 more 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. For more information, see Section 3.6.3, "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 more information about the different object scopes, see Section 5.6, "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.

3.6.2 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. Example 3–22 shows the code added to the `MyBean` class stored in the view package.

Example 3–22 Generated Code for a Managed Bean

```
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. Example 3–23 shows the `managed-bean` element created for the `MyBean` class.

Example 3–23 Managed Bean Configuration on the faces-config.xml File

```
<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>
```

3.6.3 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. For more information, see the "Using a Managed Bean in a Fusion Web Application" section 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. Example 3–24 shows how a managed bean might use the `UIComponentReference` API to get and set values for a search field.

Example 3–24 Session Scoped Managed Bean Uses the UIComponentReference API

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

3.7 Viewing ADF Faces Javadoc

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.

3.7.1 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 contains the following chapters:

4

Using ADF Faces Client-Side Architecture

This chapter outlines the ADF Faces client-side architecture.

This chapter includes the following sections:

4.1 About Using ADF Faces Architecture

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. Section 4.3, "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 Section 4.7, "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, Section 4.8, "Locating a Client Component on a Page," explains how to use the API to locate a specific client-side component, and Section 4.5, "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. For more information, see Section 4.9, "JavaScript Library Partitioning."

4.2 Adding JavaScript to a Page

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:** Including JavaScript only in the pages that need it will result in better performance because those pages that do not need it will not have to load it, as they would if the JavaScript were included in a template. However, if you find that most of your pages use the same JavaScript code, you may want to consider including the script or the tag to import the library in a template.
>
> Note, however, that 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 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.

4.2.1 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 more information, see Section 4.2, "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 in Example 4–1.

Example 4–1 MyFaces Trinidad Tag Library on a Page

```
<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, the `sayHello` function shown in Example 4–6 might be included in a JSF page as shown in Example 4–2.

Example 4–2 Inline JavaScript

```
<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.

4.2.2 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 more information, see Section 4.2, "Adding JavaScript to a Page."

To access a JavaScript library from a page:

1. Below the `document` tag, add the code shown in bold in Example 4–3 and replace `/mySourceDirectory` with the relative path to the directory that holds the JavaScript library.

Example 4–3 Accessing a 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.

4.2.3 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. Example 4–4 shows the `sayHello` function accessing the source client component in order to display its name.

Example 4–4 Accessing a Client Event Source

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

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

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

For more information about accessing client event sources, see Section 6.3, "Using JavaScript for ADF Faces Client Events." For more information about accessing client-side properties, see Section 4.5, "Accessing Component Properties on the Client." For a complete description of how client events are handled at runtime, see Section 6.3.7, "What Happens at Runtime: How Client-Side Events Work."

4.3 Instantiating Client-Side Components

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`.

4.3.1 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. For more information, see Section 4.3, "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`.

4.3.2 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. Example 4–5 shows the 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.

Example 4–5 Adding a Component

```
<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.

4.4 Listening for Client Events

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.

4.4.1 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. For more information, see Section 4.4, "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 in Example 4–6.

Example 4–6 JavaScript Event Handler

```
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. Example 4–7 shows the code that would be created for the listener for the `sayHello` function.

Example 4–7 Registering a Client Listener

```
<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 more information about client-side events, see Section 6.3, "Using JavaScript for ADF Faces Client Events."

4.5 Accessing Component Properties on the Client

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 rerendered 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 `commandLink` 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 more information about using the `immediate` property, see Section 5.2, "Using the Immediate Attribute"). Consequently, the `immediate` property is a secured property.

Attempts to set secured property from JavaScript will fail. For more information, see Section 4.5.3, "How to Unsecure the disabled Property." Table 4–1 shows the secure properties on the client components.

Table 4–1 Secure Client Properties

Component	Secure Property
`AdfRichChooseColor`	`colorData`
`AdfRichComboboxListOfValue`	`disabled` `readOnly`

`AdfRichCommandButton`	`disabled`
	`readOnly`
	`blocking`

`AdfRichCommandImageLink`	`blocking`
	`disabled`
	`partialSubmit`

`AdfRichCommandLink`	`readOnly`
`AdfRichDialog`	`dialogListener`
`AdfRichDocument`	`failedConnectionText`

`AdfRichInputColor`	`disabled`
	`readOnly`
	`colorData`

Table 4–1 (Cont.) Secure Client Properties

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

Table 4–1 (Cont.) Secure Client Properties

Component	Secure Property
AdfRichSelectManyListBox	disabled readOnly valuePassThru
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

Table 4–1 (Cont.) Secure Client Properties

Component	Secure Property
`AdfUIPage`	`rowDisclosureListener` `immediate`
`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`

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.

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.

4.5.1 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*). Example 4–8 shows how you might use the `setProperty` function to set the `backgroundcolor` property on an `inputText` component to red when the value changes.

Example 4–8

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

4.5.2 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.

4.5.3 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, he 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.

4.5.4 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, as shown in

Example 4–9 Using JavaScript to Set the changed Property

```
<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. Example 4–10 shows example code.

Example 4–10 JSF Code for Setting Property Values on the Server

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

Example 4–11 shows the corresponding managed bean code.

Example 4–11 Using a Managed Bean to Set a Property Value

```
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. Use this when there is no event being delivered. Example 4–12 shows the JSF code.

Example 4–12 JSF Code for Setting Property Values Using JavaScript and a Server Listener

```
<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>
```

Example 4–13 shows the managed bean code.

Example 4–13 Using a Custom Event to Set a Property Value

```
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`. Example 4–14 shows the JSF code.

Example 4–14 JSF Code for Using a Listener on a Command Component to Set a Property Value

```
<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>
```

Example 4–15 shows the corresponding managed bean code.

Example 4–15 Using an ActionLIstener to Set a Property Value

```
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;
}
```

4.6 Using Bonus Attributes for Client-Side Components

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 more information on marshalling, see Section 6.4.3, "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`. For more information, see Section 4.3, "Instantiating Client-Side Components."

4.6.1 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. For more information, see Section 4.6, "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.

4.6.2 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. For more information, see Section 6.4, "Sending Custom Events from the Client to the Server."

4.7 Understanding Rendering and Visibility

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.

Example 4–16 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.

Example 4–16 Rendered and Not Rendered Components

```
<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.

Example 4–17 shows how you might achieve the same functionality as shown in Example 4–16, 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).

Example 4–17 Visible and Not Visible Components

```
<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.

Example 4–18 shows the page code for JavaScript that handles the visiblity of the components.

Example 4–18 Using JavaScript to Turn On Visibility

```
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)
    }

 }
```

4.7.1 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. For more information, see Section 4.7, "Understanding Rendering and Visibility."

To set visibility:

1. Create the JavaScript that can toggle the visibility. Example 4–18 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`.

Example 4–19 shows the full page code used to toggle visibility with JavaScript.

Example 4–19 JavaScript Toggles 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>
```

4.7.2 What You May Need to Know About Visible and the isShowing 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`.

4.8 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.

Example 4–20 shows the `sayHello` function finding the `outputText` component using the component's ID.

Example 4–20 Finding a Client Component Using findComponent()

```
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.

4.8.1 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. For more information, see the *Java API Reference for Oracle ADF Faces*.

4.9 JavaScript Library Partitioning

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. When you create custom ADF Faces components, you can create your own features and partition configuration files for those components.

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. For more information, see Section A.2.3.17, "JavaScript Partitioning."

4.9.1 How to Create a JavaScript Feature

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

> **Note:** You create JavaScript features when you create custom ADF Faces components. All existing ADF Faces components already have features created for them, and these cannot be changed.

Before you begin

It may be helpful to have an understanding of JavaScript partitioning works. For more information, see Section 4.9, "JavaScript Library Partitioning."

If not already created, create a `META-INF` directory for your component.

To create a JavaScript feature:

1. In the Applications window, right-click **META-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-features.xml` as the file name and save it in the `META-INF` directory.
4. In the source editor, replace the generated code with the code shown in Example 4–21.

Example 4–21 XML for adf-js-features.xml File

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

</adf-js-features>
```

5. Add the following elements to populate a feature with the relevant component files and dependencies.
   - `features`: The root element of the configuration file.
   - `feature`: Create as a child to the `features` element. This element must contain one `feature-name` child element and can also contain any number of `feature-class`, as well as any number of `feature-dependency` elements.
   - `feature-name`: Create as a child to the `feature` element. Specifies the name of the feature. You must use the client component's constructor name for this value.
   - `feature-class`: Create as a child to the `feature` element. Specifies the location of the single JavaScript file or class to be included in this feature. There can be multiple `feature-class` elements.
   - `feature-dependency`: Create as a child to the `feature` element. Specifies the name of another feature that this feature depends on. For example, if one component B extends component A, then the feature that represents component A must be listed as a dependency for component B. By noting dependencies, the framework can ensure that any dependent classes are available, even if the two features are not in the same partition.

   Example 4–22 shows the `feature` element for a fictitious custom component that uses popup components (and therefore has a dependency to the popup feature).

Example 4–22 JavaScript Features Configuration

```
<features xmlns="http://xmlns.oracle.com/adf/faces/feature">
  <feature>
    <feature-name>AcmeMyPane</feature-name>
    <feature-class>
      oracle/adfdemo/acme/js/component/AcmeMyPane.js
    </feature-class>
    <feature-class>
      oracle/adfdemo/acme/js/event/AcmePaneSelectEvent.js
    </feature-class>
    <feature-class>
      oracle/adfdemo/acme/js/component/AcmeMyPanePeer.js
    </feature-class>

  <!-- Dependencies -->

  <!-- Popup hints -->
  <feature-dependency>AdfRichPopup</feature-dependency>

</feature>
```

4.9.2 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 Section F.1.1, "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 more information, see Section 4.9, "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 code shown in Example 4–23.

Example 4–23 XML for adf-js-partitions.xml File

```
<?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.

Example 4–24 shows the `partition` element for the `tree` partition that contains the `AdfRichTree` and `AdfRichTreeTable` features.

Example 4–24 JavaScript Partition Configuration

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

4.9.3 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:

5.1 About Using the JSF Lifecycle and ADF Faces

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.

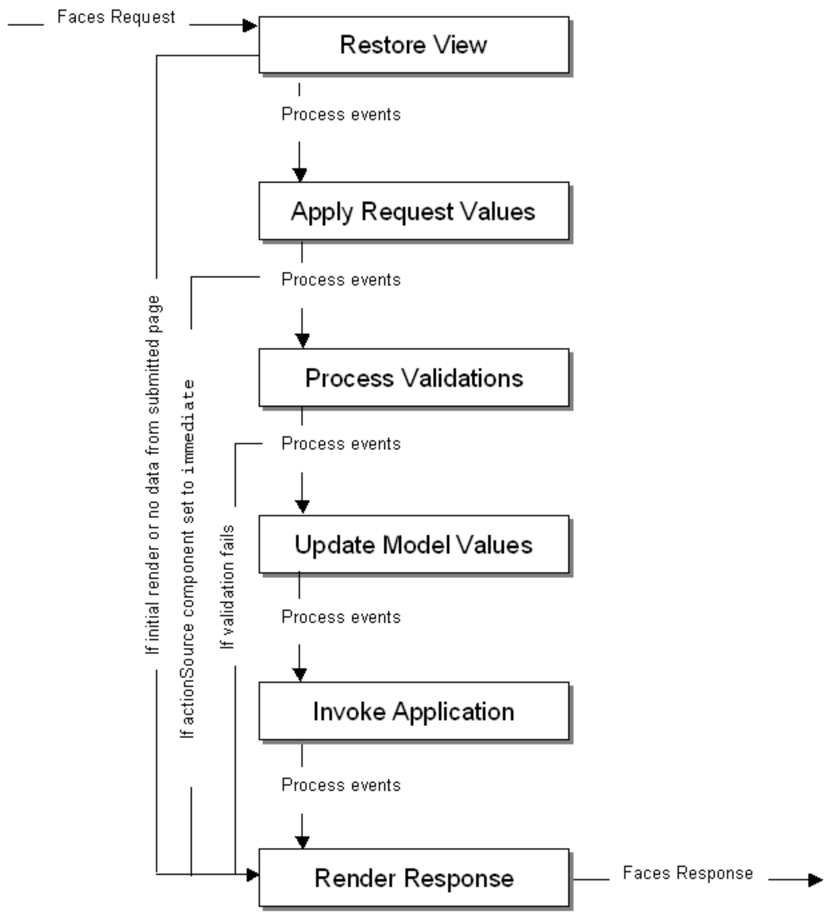
To better understand the lifecycle enhancements that the framework delivers, it is important that you understand the standard JSF lifecycle. This section provides only an overview. For a more detailed explanation, refer to the JSF specification at `http://www.jcp.org/en/jsr/detail?id=314`.

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. For more information, see Section 5.2, "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 more information about conversion and validation, see Chapter 7, "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.

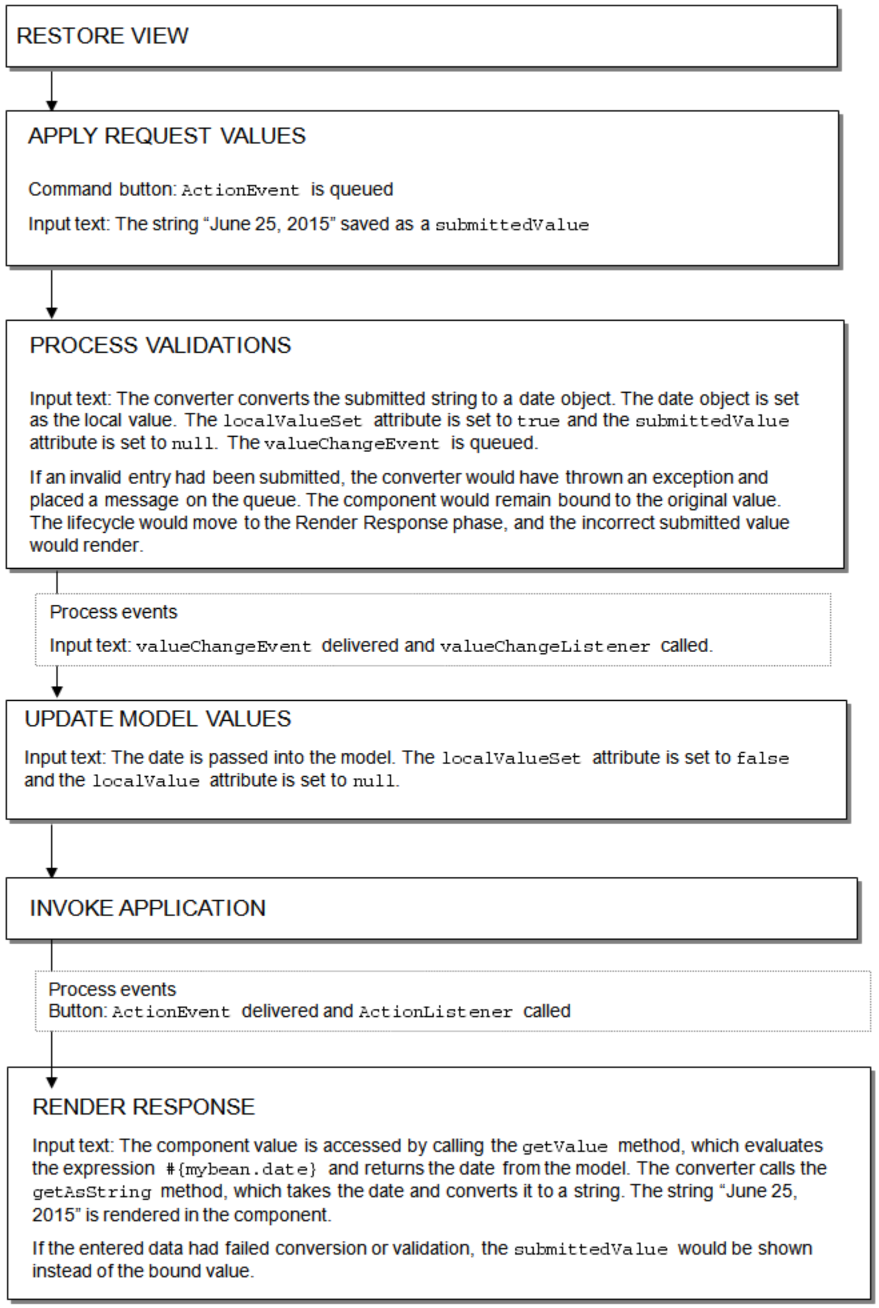
To help illustrate the lifecycle, consider a page that has a simple input text component where a user can enter a date and then click a button to submit the entered value. A `valueChangeListener` method is also registered on the component. Example 5–1 shows the code for the example.

Example 5–1 Sample Code to Illustrate the JSF Lifecycle

```
<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



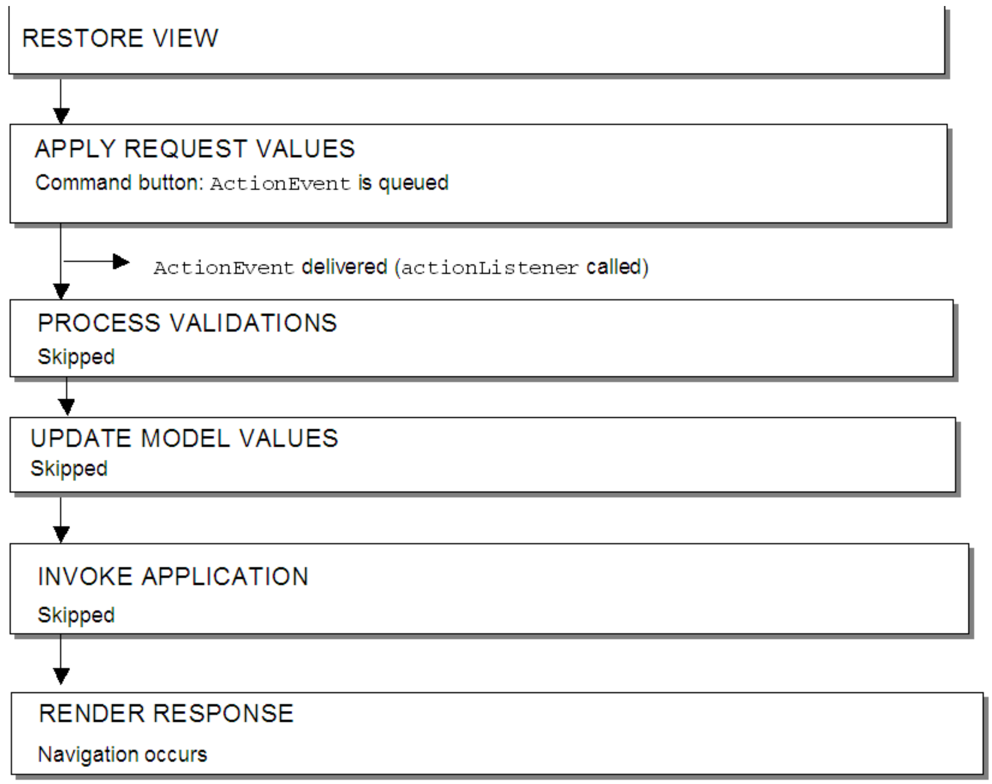
5.2 Using the Immediate Attribute

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 `button`) 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.

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 more information about navigation, see Chapter 20, "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

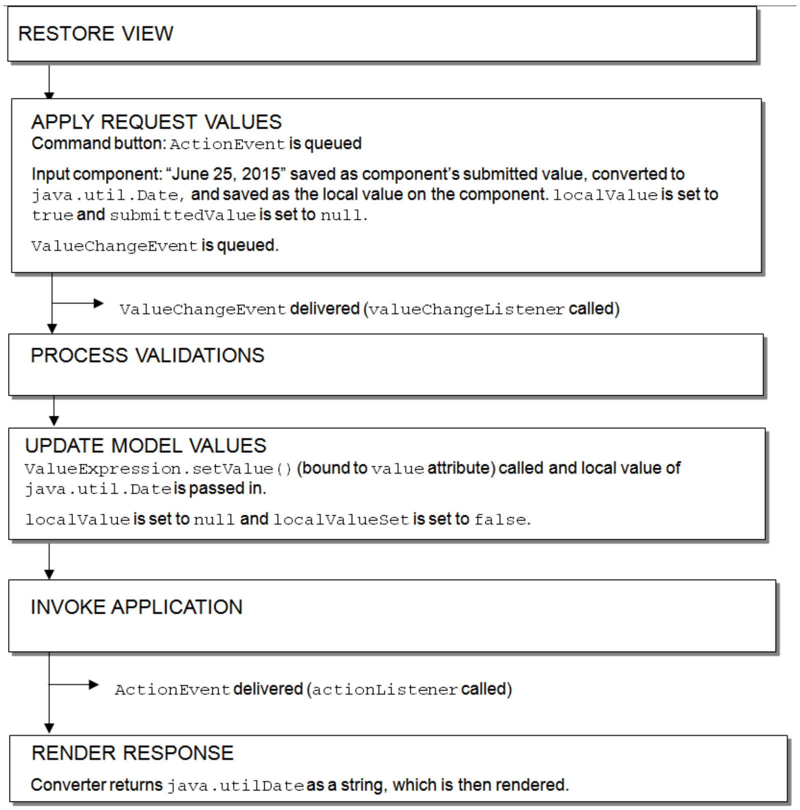


Note: A button that does not provide any navigation and is set to `immediate` will also go directly to the Render Response phase: the Validation, Update Model, and Invoke Application phases are skipped, so any new values will not be pushed to the server.

You can also set components that invoke disclosure events (such as a `showDetail` component), and components that are `editableValueHolder` components (components that hold values that can change, such as an `inputText` component), to immediate. As with `actionSource` components, the events are then delivered to the Apply Request Values phase. However, for `editableValueHolder` components, instead of skipping phases, conversion, validation, and delivery of `valueChangeEvents` events 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 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.

> **Performance Tip:** 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 `commandNavigationItem` 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 Section 20.9.1, "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.

As another example, suppose you have a form with an input component used to search for a string, with a button configured to invoke the search execution, and another input text component used to enter a date, with an associated button used to submit the date. In this example, you want to set the search input component and its button both to be `immediate`. This will allow the user to execute a search, even if an invalid string is entered into the date field, because the date input component’s converter is never fired. Also, because the search input text is set to `immediate` and the date input field is not, only the search input text will be processed. And because both fields are within the same form, if the user enters a valid date in the date field, but then performs a search and does not click the **Save** button, the entered value will still be displayed when the search results are displayed. Example 5–2 shows the code used for the two fields and two buttons.

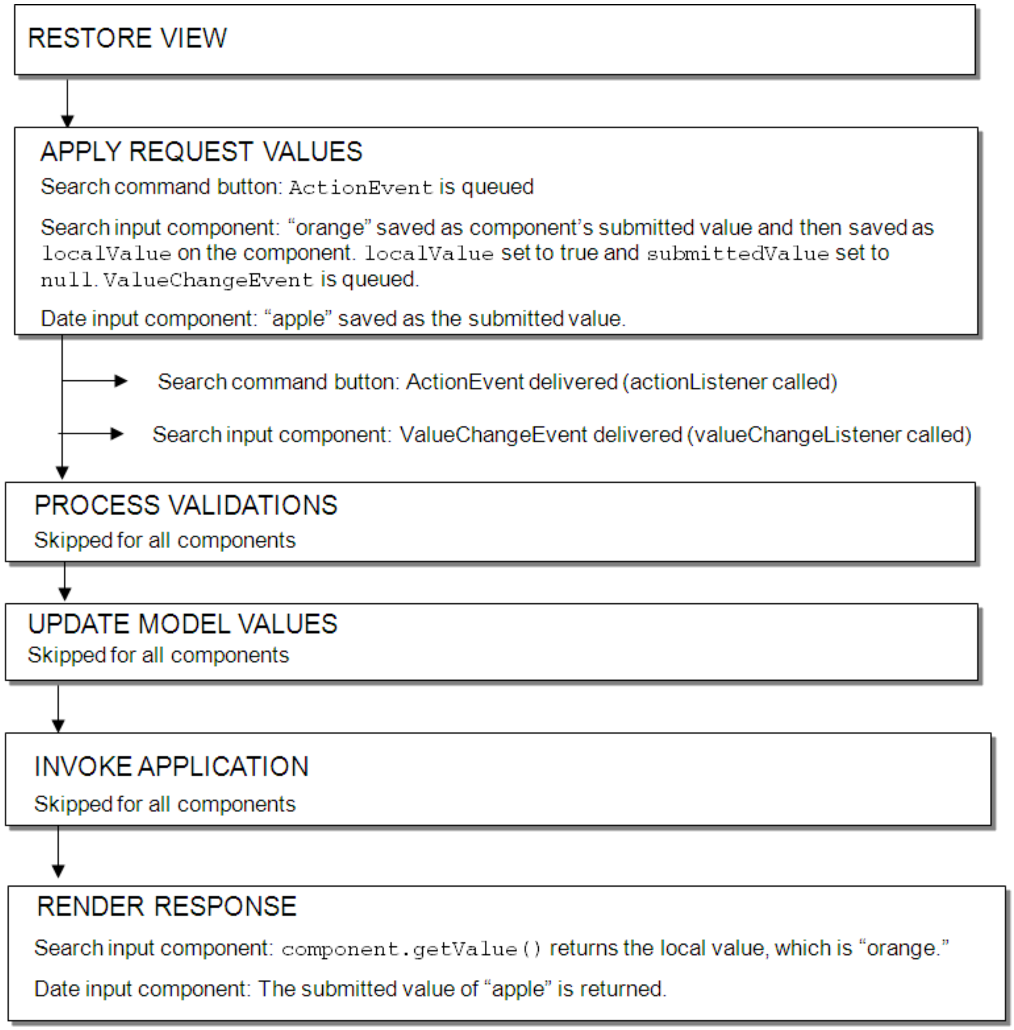
Example 5–2   Input Component and Command Components Using Immediate

```
<af:form>
  <af:inputText immediate="true" label="Search"  value="#{mybean.search}"
                valueChangeListener="#{mybean.searchValueChangeListener}"/>
  <af:button immediate="true" text="search"
                    actionListener="#{mybean.searchActionListener}"/>
  [.... tags to render search result ....]

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

Figure 5–5 shows the lifecycle for this page when a user does the following:

- Enters `apple` into the **Date** input field (which is not a valid entry)
- Enters `orange` into the **Search** field
- Clicks the **Search** button to execute the search on `orange`
- Clicks the **Save** button to save the value `apple` as the date

Figure 5–5 Immediate Attribute on Both Command Component and Input Component



When using the `immediate` attribute for `editableValueHolder` and `actionSource` components on the same page, note the following issues:

- If an `actionSource` component requires data from an `editableValueHolder` component, that data will not be available to the model until after the Update Model Values phase. If you have an immediate `actionSource` component, and that component needs data, then set `immediate` on the `editableValueHolder` fields as well. Then, you can call the `getValue` method on the `editableValueHolder` component and the local value will be returned. It will not have been pushed into the model yet, but it will be available on the component.
- If an immediate `editableValueHolder` component fails validation, any immediate `actionSource` component will still execute.

5.2.1 How to Use the Immediate Attribute

Before you begin

It may be helpful to have an understanding of the `immediate` attribute. For more information, see Section 5.2, "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`.

5.3 Using the Optimized Lifecycle

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. An event root component can be decided in two ways:

- Components: Certain components are always event root components. Regions and popups are an examples of a component which the framework knows is a boundary. No matter what event is triggered inside a region or popup, the lifecycle does not run on components outside the region or popup.
- Events: Certain events indicate a component as a root. For example, the disclosure event sent when expanding or collapsing a `showDetail` component (see Section 9.9, "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. Table 6–1 in Section 6.1.1, "Events and Partial Page Rendering" shows the events that have a corresponding event root component.

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 more information about how the ADF Faces framework uses PPR, and how you can use PPR throughout your application, see Chapter 8, "Rerendering Partial Page Content."

5.4 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 more information about subforms, see Section 5.5, "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.

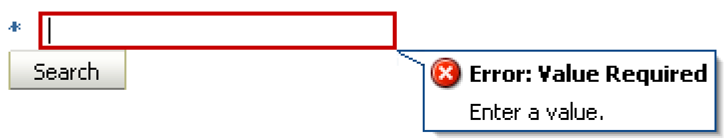
For example, ADF Faces provides the `required` attribute for input components, and this validation runs on the client. When you set this attribute to `true`, the framework will show an error on the page if the value of the component is `null`, without requiring a trip to the server. Example 5–3 shows 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.

Example 5–3 Simple Client-Side Validation Example

```
<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 5–6. There will be no trip to the server; this error detection and message generation is all done on the client.

Figure 5–6 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 more information about using client-side validation and conversion, see Chapter 7, "Validating and Converting Input."

5.5 Using Subforms to Create Sections on a Page

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 more information about subforms, see Section 11.2, "Defining Forms."

5.6 Object Scope Lifecycles

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:

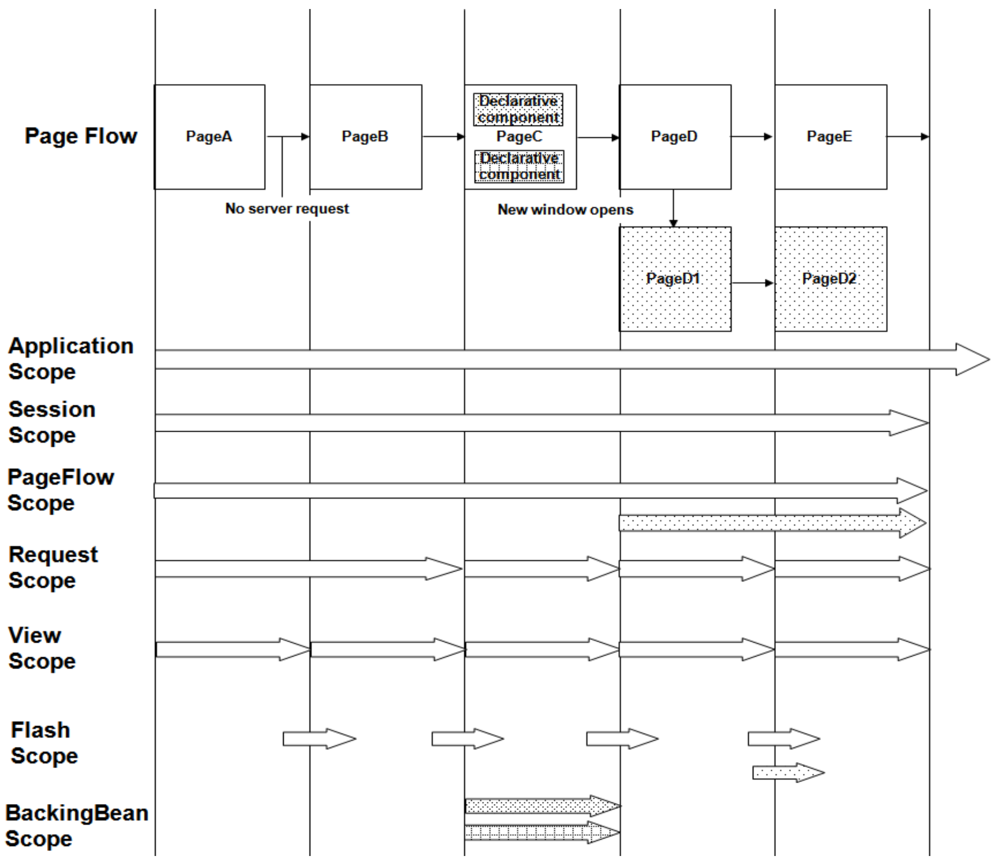
- `pageFlowScope`: The object is available as long as the user continues navigating from one page to another. If the user opens a new browser window and begins navigating, that series of windows will have its own `pageFlowScope` scope.
- `backingBeanScope`: Used for managed beans for page fragments and 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 5–7 shows the time period in which each type of scope is valid, and its relationship with the page flow.

Figure 5–7 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. For more information, see the "Using a Managed Bean in a Fusion Web Application" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

5.7 Passing Values Between Pages

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 the "Getting Started with ADF Task Flows" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

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.

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.

5.7.1 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 more information, see Section 5.6, "Object Scope Lifecycles." You may also want to understand how pageFlow scope is used to pass values. For more information, see Section 5.7, "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();
   ```

5.7.2 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 more information about creating command components, see Section 20.3, "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.

To access a value from the pageFlowScope scope:

1. On the page from which you want to access the value, drop the component that you want to display the value.
2. 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}`.

5.7.3 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:

6.1 About Events and Event Handling

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

6.1.1 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.

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 more information about this aspect of the lifecycle, see Section 5.3, "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 rerenderd, and not the whole page.

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 Section 9.9, "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.

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

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

Event Type	Component Trigger	Is Event Root
`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`
`valueChange`	All input and select components (components that implement `EditableValueHolder`)	`true`

> **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. For more information, see Chapter 8, "Rerendering Partial Page Content."

6.1.2 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.

6.2 Using ADF Faces Server 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. For more information, see Chapter 20, "Working with Navigation Components."
`ActiveDataEvent`	Used to update components based on events. For more information see the "Using the Active Data Service" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.
`AttributeChangeEvent`	All input and select components (components that implement `EditableValueHolder`). For more information, see Chapter 11, "Using Input Components and Defining Forms."

`CalendarActivity DurationChangeEvent`
`CalendarActivityEvent`
`CalendarDisplay ChangeEvent`
`CalendarEvent`
	The Calendar component. For more information, see Chapter 17, "Using a Calendar Component."

`CarouselSpinEvent`	The `carousel` component. For more information, see Section 12.11, "Displaying Images in a Carousel."
`ColumnSelectionEvent` `ColumnVisibility ChangeEvent`	The `table` and `treeTable` components. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."
`ContextInfoEvent`	The `contextInfo` component. For more information, see Section 15.5, "Displaying Contextual Information in Popups."
`DialogEvent`	The `dialog` component. For more information, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."
`DisclosureEvent`	`The showDetail`, `showDetailHeader`, `showDetailItem` components. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically" and Section 9.10, "Displaying or Hiding Contents in Panels."
`DropEvent`	Components that support drag and drop. For more information, see Chapter 36, "Adding Drag and Drop Functionality."
`FocusEvent` *	The tree and `treeTable` components. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."
`ItemEvent`	The `panelTabbed` component. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels." Also, the navigationPane component. For more information, see Section 20.6, "Using Navigation Items for a Page Hierarchy."
`LaunchEvent`	All command components. For more information, see Chapter 20, "Working with Navigation Components."
`LaunchPopupEvent`	The `inputListOfValues` and `inputComboboxListOfValues` components. For more information, see Chapter 14, "Using Query Components."
`LoadEvent` **	The `document` component. For more information, see Section 9.2.5, "How to Configure the document Tag."

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

Event	Triggered by Component...
`PollEvent`	The `poll` component. For more information, see Section 6.7, "Using Polling Events to Update Pages"
`PopupCanceledEvent` `PopupFetchEvent`	The popup component. For more information, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."
`QueryEvent` `QueryOperationEvent`	The `query` and `quickQuery` components. For more information, see Chapter 14, "Using Query Components."
`RangeChangeEvent`	The `table` component. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."
`RegionNavigationEvent`	`The region` component. For more information, see the "Using Task Flows as Regions" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.
`ReturnEvent`	All command components. For more information, see Chapter 20, "Working with Navigation Components."
`ReturnPopupEvent`	The `inputListOfValues` and `inputComboboxListOfValues` components. For more information, see Chapter 14, "Using Query Components."
`ReturnPopupDataEvent`	The popup component. For more information, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."
`RowDisclosureEvent`	The tree and `treeTable` components, as well as the `treemap` and `sunburst` DVT components. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components" and Chapter 30, "Using Treemap and Sunburst Components."
`SelectionEvent`	The `table`, `tree`, and `treeTable` components, as well as the `treemap` and `sunburst` DVT components. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components" and Chapter 30, "Using Treemap and Sunburst Components."
`SortEvent`	The `table` and `treeTable` components. For more information, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."
`ValueChangeEvent`	All input and select components (components that implement `EditableValueHolder`). For more information, see Chapter 11, "Using Input Components and Defining Forms."
`WindowLifecycleEvent`	Delivered when the `LifecycleState` of a window changes. For more information, see the *Java API Reference for Oracle ADF Faces*.
`WindowLifecycle NavigateEvent`	Delivered when the current window is unloaded in order to navigate to a new location. For more information, see the *Java API Reference for Oracle ADF Faces*.

* 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).

6.2.1 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. For more information, see Section 6.2, "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`. Example 6–1 shows the code for the `tableSelectFileItem` handler. (For information about creating and using managed beans, see Section 3.6, "Creating and Using Managed Beans.")

Example 6–1 Event Listener Method

```
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.

   Example 6–2 shows sample code for registering a selection event listener method on a `table` component.

Example 6–2 Registering an Event Listener Method

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

6.3 Using JavaScript for ADF Faces Client Events

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. For more information, see Section 6.3.5, "How to Prevent Events from Propagating to the Server."

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
`AdfActionEvent`	action	Yes	Yes	All command components
`AdfBusyStateEvent`	`busyState`	No	No	Triggered by the page
`AdfCarouselSpinEvent`	`event`	Yes	No	`carousel`
`AdfColumnSelectionEvent`	`event`	Yes	Yes	`table`, `treeTable`
`AdfComponentEvent`	load	Yes	Yes	`document` After the document's contents have been displayed on the client, even when PPR navigation is used. It does not always correspond to the onLoad DOM event.
`AdfComponentFocusEvent`		No	Yes	Any component that can receive focus
`AdfDialogEvent`	event	Yes	Yes	`dialog` When user selects the OK or Cancel button in a dialog

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

Event Class	Event Type	Propagates to Server	Can Be Canceled	Triggered by Component
AdfDisclosureEvent	event	Yes	Yes	panelBox, region, showDetail, showDetailHeader, showDetailItem When the disclosure state is toggled by the user
AdfDomComponentEvent	inlineFrameLoad	Yes	Yes	inlineFrame When the internal iframe fires its load event.
AdfDropEvent	drop	Yes	No	Any component that supports drag and drop
AdfFocusEvent	focus	Yes	Yes	tree, treeTable
AdfItemEvent	item	Yes	Yes	commandNavigationItem showDetailItem
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

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

Event Class	Event Type	Propagates to Server	Can Be Canceled	Triggered by Component
`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`
`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

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. Example 6–3 shows an example of a JavaScript function associated with an action event.

Example 6–3 clientListener Tag

```
<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, Example 6–4 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.

Example 6–4 Using the f:ajax Client Behavior Tag

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

6.3.1 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. For more information, see Section 6.3, "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 Section 4.2, "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 more information about locating client components, see Section 4.8, "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. For more information, see Section 6.3.2, "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 marshall custom server-side attributes to the client. For more information, see Section 6.3.3, "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. For more information, see Section 6.3.5, "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. For more information, 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.

Example 6–5 shows the code used to invoke the `showHelpFileExplorerPopup` function from the `Explorer.js` JavaScript file.

Example 6–5 clientListener Tags on JSF Page

```
<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. Example 6–6 shows the code used to set attribute values for the `showAboutFileExplorerPopup` function.

Example 6–6 Adding Attributes

```
<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.

6.3.2 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 Example 6–7, 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 Section 6.3.3, "How to Use Client-Side Attributes for an Event").

Example 6–7 Finding the Source Component of a Client 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.

6.3.3 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 Example 6–8, 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.

Example 6–8 Attribute Values Are Accessed from JavaScript

```
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 Example 6–9.

Example 6–9 Setting Attributes on a Component

```
<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.

6.3.4 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. Example 6–10 shows how you can use `preventUserInput` in your JavaScript.

Example 6–10 Blocking UI Input

```
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);
}
```

6.3.5 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`.

Example 6–11 shows the `showAboutFileExplorerPopup` function, which cancels its propagation.

Example 6–11 Canceling a Client Event from Propagating to the Server

```
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 uncancel an event once it has been canceled.

6.3.6 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 r the call to return To let the framework know that no response is expected, you use the `AdfBaseEvent.noResponseExpected()` method.

6.3.7 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 more information about peers and the ADF Faces architecture, see Chapter 4, "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: Canceling 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.

6.3.8 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 Example 6–12. The `getSource()` method accesses the `AdfUIComponent` class, which can then be used to find the component.

Example 6–12 JavaScript Using the findComponent() Method

```
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 more information about working with naming containers, see Section 4.8, "Locating a Client Component on a Page."

6.4 Sending Custom Events from the Client to the Server

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 Example 6–13 shows, this happens because the `inputText` field contains a `clientListener` that invokes a JavaScript function when the **Enter** key is pressed.

Example 6–13 clientListener Invokes JavaScript Function and Causes ServerLIstener to Be Invoked

```
//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.

6.4.1 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. For more information, see Section 6.4, "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 Example 6–14.

Example 6–14 Sample JavaScript for Custom Events

```
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. Example 6–15 shows the code used in the `SearchNavigatorView` managed bean that simply calls another method to execute the search and then refreshes the navigator.

Example 6–15 Server Listener Method for a Custom Client Event

```
  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 Section 6.4.3, "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.

6.4.2 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.

6.4.3 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.

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`	`java.lang.Boolean`
`Number`	`java.lang.Double`
`String`	`java.lang.String`
`Date`	`java.util.Date`
`Array`	`java.util.ArrayList`
`Object`	`java.util.Map`

Table 6–6 shows how Java types map back to JavaScript types.

Table 6–6 Java to JavaScript Type Map

Java Type	JavaScript Type
`java.lang.Boolean`	`Boolean`
`java.lang.Double`	`Number`
`java.lang.Integer`	`Number`
`java.lang.Float`	`Number`

Table 6–6 (Cont.) Java to JavaScript Type Map

Java Type	JavaScript Type
java.lang.Long	Number
java.lang.Short	Number
java.lang.Character	String
java.lang.CharSequence	String
java.util.Collection	Array
java.util.Date	Date
java.util.Map	Object
Array	Array
java.awt.Color	TrColor

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`.

6.5 Executing a Script Within an Event Response

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.

Example 6–16 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.

Example 6–16 Invoking a Method to Add JavaScript to a Response

```
<af:btton id="upOneFolder"
. . .
        actionListener="#{explorer.headerManager.handleUpOneFolder}"/>
```

Example 6–17 shows the `handleUpOneFolder` method that uses the `ExtendedRenderKitService` class.

Example 6–17 Adding JavaScript to a Response

```
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();
    }

  }
```

6.6 Using ADF Faces Client Behavior Tags

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 `panelDasboard` component to appear more responsive. For details, see Section 9.8.1, "How to Use the panelDashboard Component."
- `insertTextBehavior`: Enables a command component to insert text at the cursor in an `inputText` component. For details, see Section 11.3.2, "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. For details, see Section 11.8.2, "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. For more information, see Section 13.1, "About List-of-Values Components."
- `showPopupBehavior`: Enables a command component to launch a popup component. For details, see Section 15.3, "Declaratively Invoking a Popup."
- `showPrintablePageBehavior`: Enables a command component to generate and display a printable version of the page. For details, see Section 37.2, "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 Chapter 20, "Working with Navigation Components."
- `scrollComponentIntoViewBehavior`: Enables a command component to jump to a named component when clicked. For details, see Section 6.6.1, "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. For more information, see the *Java API Reference for Oracle ADF Faces*.

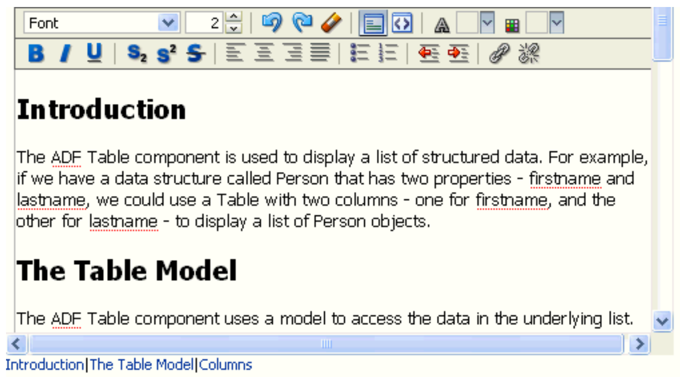
- `target`: Enables a component to declaratively execute or render a list of components when a specified event occurs. For details, see Section 8.3, "Using the Target Tag to Execute PPR."

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 Section 6.3, "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 Section 6.4, "Sending Custom Events from the Client to the Server").

6.6.1 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. For more information, see Section 6.6, "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 Section 20.3.1, "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 Example 6–18. Note that `Introduction` is the ID for the first header.

Example 6–18 subTargetId Value Defined in a Property

```
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>";
```

6.7 Using Polling Events to Update Pages

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 (for more information, see Section A.2.3.24, "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.

6.7.1 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. For more information, see Section 6.7, "Using Polling Events to Update Pages."

To use a poll component:

1. In a managed bean, create a handler for the poll event. For more information about managed beans, see Section 3.6, "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:

7.1 About ADF Faces Converters and Validators

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 more information about displaying messages in an ADF Faces application, see Chapter 19, "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.

7.1.1 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.

7.1.2 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 Chapter 5, "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 more information about creating messages, see Chapter 19, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."

7.2 Conversion, Validation, and the JSF Lifecycle

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 Chapter 5, "Using the JSF Lifecycle with ADF Faces."

7.3 Adding Conversion

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 `java.lang.String` objects to `java.awt.Color` objects. You specify a set of color patterns as an attribute of the converter.

Table 7–1 (Cont.) ADF Faces Converters

Converter	Tag Name	Description
`DateTimeConverter`	`af:convertDateTime`	Converts `java.lang.String` objects to `java.util.Date` objects. You specify the pattern and style of the date as attributes of the converter.
`NumberConverter`	`af:convertNumber`	Converts `java.lang.String` objects to `java.lang.Number` 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.

7.3.1 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. For more information, see Section 7.3, "Adding Conversion."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 7.1.2, "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. For more information, see Section 7.3.2, "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 more information about creating messages, see Chapter 19, "Displaying Tips, Messages, and Help."

7.3.2 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.

Example 7–1 illustrates the use of a multiple pattern for the `af:convertColor` tag in which "255-255-000" and "FFFF00" are both acceptable values.

Example 7–1 af:convertColor Multiple Patterns

```
<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>
```

Example 7–2 illustrates the use of an `af:convertDateTime` tag in which "6/9/2007" and "2007/9/6" are both acceptable values.

Example 7–2 af:convertDateTime Multiple Patterns

```
  <af:inputDate id="mdf5" value="2004/09/06" label="attached converter">
    <af:convertDateTime pattern="yyyy/M/d" secondaryPattern="d/M/yyyy" />
  </af:inputDate>
```

Example 7–3 illustrates an `af:convertNumber` tag with the `type` attribute set to `currency` to accept "$78.57" and "$078.57" as values for conversion.

Example 7–3 af:convertNumber Set to Currency Attribute

```
<af:inputText label="type=currency" value="#{validate.currency}" id="it1">
  <af:convertNumber type="currency"/>
</af:inputText>
```

7.3.3 What Happens at Runtime: How Converters Work

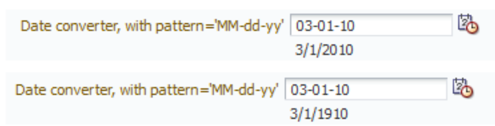
When the user submits the page containing converters, the ADF Faces `validate()` method calls the converter's `getAsObject()` method to convert the `String` value to the required object type. When there is not an attached converter and if the component is bound to a bean property in the model, then ADF 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.

7.3.4 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

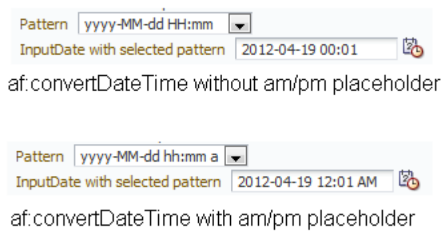


If you are using a two-digit year format, all strings containing two-digit years will be resolved into a date within two-digit-year-start and two-digit-year-start + 100. For example, if two-digit-year-start value is 1912, the string 01/01/50 gets resolved to 01/01/1950. To enter dates outside this range, the end user should enter a date with the full (four-digit) year. For more information about two-digit-year-start element and how to configure it, see Section A.6.2, "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)`.

7.4 Creating Custom ADF Faces 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.

7.4.1 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.

7.4.1.1 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 more 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 more 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. For more information, see the Javadoc for the `StateHolder` interface of `javax.faces.component` package, or visit http://docs.oracle.com/javaee/index.html.

7.4.1.2 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.

7.4.1.3 Implement Client-Side (Javascript) Conversion

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.

7.4.1.3.1 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 Example 7–4.

Example 7–4 Interface Converter

```
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.

Example 7–5 shows the signature for `TrFacesMessage`.

Example 7–5 TrFacesMessage Signature

```
/**
 * 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){
  ..
}
```

Example 7–6 shows the signature for `TrFacesException`.

Example 7–6 TrFacesException Signature

```
/**
* 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){
  ..
}
```

7.4.1.3.2 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 more information on the interface methods, see the `ClientConverter` javadoc on `http://myfaces.apache.org`.

Example 7–7 shows a custom javascript converter implementation for social security number.

Example 7–7 Custom Client-Side Converter in JavaScript

```
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';
```

Example 7–8 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.

Example 7–8 Custom Server-Side Converter in Java

```
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';";

}
```

7.4.2 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:

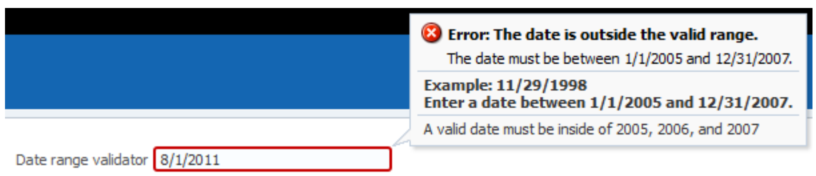
```
<af:inputText value="#{myBean.myProperty}"/>
```

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.

7.5 Adding Validation

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. For information, see Section 7.5.1.2, "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 Section 7.5.1.3, "Using ADF Faces Validators." You can also create your own validators. For information about custom validators, see Section 7.6.3, "How to Create a Custom JSF Validator."

7.5.1 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.

7.5.1.1 Adding ADF Faces Validation

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. For more information, see Section 7.5.1.2, "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. For more information, see Section 7.5.1.3, "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. For more information, see

Section 7.6, "Creating Custom JSF Validation."

When validation is added, validation errors can be displayed inline or in a popup window on the page. For more information about displaying messages created by validation errors, see Chapter 19, "Displaying Tips, Messages, and Help."

7.5.1.2 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**.

7.5.1.3 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 `maximumLength` attribute of `inputText` 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 `java.lang.String`.
`LongRangeValidator`	`af:validateLongRange`	Validates that a component value is within a specified range. The value must be any numeric type or `String` 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 Section 7.6, "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.

   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.

7.5.2 What Happens at Runtime: How Validators Work

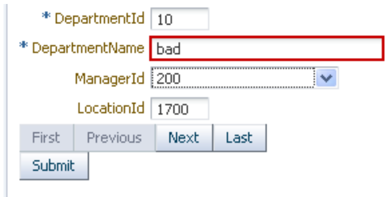
When the user submits the page, ADF Faces checks the submitted value and runs conversion on any non-null value. The converted value is then passed to the `validate()` method. If the value is empty, the `required` attribute of the component is checked and an error message is generated if indicated. If 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 redisplays 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



For more information about adding error messages when a validation or conversion error occurs, see Section 19.3, "Displaying Hints and Error Messages for Validation and Conversion."

7.5.3 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. For more information see Chapter 5, "Using the JSF Lifecycle with ADF Faces."

7.6 Creating Custom JSF Validation

You can add your own validation logic to meet your specific business needs. 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.

7.6.1 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. For more information, see Section 7.6, "Creating Custom JSF Validation."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 7.1.2, "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 more 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`.

7.6.2 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. Example 7–9 shows the code JDeveloper generates.

Example 7–9 Managed Bean Code for a Validation Method

```
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.

7.6.3 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. For more information, see Section 7.6, "Creating Custom JSF Validation."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 7.1.2, "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 more 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 more 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.

   For more information, 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.

   Example 7–10 shows a validator in Java.

Example 7–10 Java Validator

```
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 Example 7–11.

Example 7–11 Client-side JavaScript Validator

```
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.

  Example 7–12 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.

Example 7–12 A Custom Validator Tag on a JSF Page

```
<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.

Example 7–13 shows the code on a JSF page for a validator using the `f:validator` tag.

Example 7–13 A Custom Validator Nested Within a Component on a JSF Page

```
<af:inputText id="empnumber" required="true">
  <f:validator validatorID="emValidator"/>
</af:inputText>
```

7.6.4 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:

8.1 About Partial Page Rendering

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 Section 9.9, "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 Table 6–1 in Section 6.1.1, "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.

Example 8–1 Example of Declarative PPR

```
<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, than 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 Example 8–2.

Example 8–2 inputText Component Within a panelGroup Component Will Be Validated with PPR

```
<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. Example 8–3 shows the JSF code to do this.

Example 8–3 Target Tag Determines the Events and Components Involved in PPR

```
<af:panelFormLayout id="pfl1">
  <af:selectBooleanRadio id="show2" autoSubmit="true" text="Show"
                         value="#{validate.show2}"/>
```

```
    <af:target events="valueChange" execute="show2 hide2" render="pfl1"/>
  <af:selectBooleanRadio id="hide2" autoSubmit="true" text="Hide"
                         value="#{validate.hide2}"/>

    <af:target events="valueChange" execute="hide2 show2" render="pfl1"/>
  <af:panelGroupLayout id="pgl1">
    <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. 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*.

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.

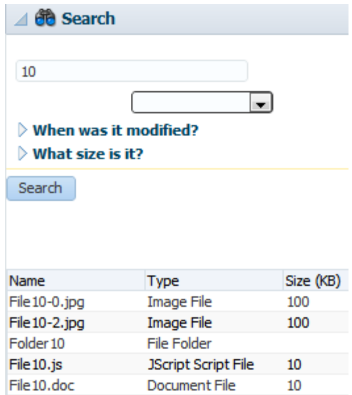
> **Note:** The browser must have JavaScript enabled for PPR to work.

8.2 Using Partial Triggers

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. For more information, see Section 8.3, "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 programatically enable PPR. For more information, see Section 8.4, "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. For more information, see Section 5.3, "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 Section 4.8, "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). For more information, see Section 8.3, "Using the Target Tag to Execute PPR."

8.2.1 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. For more information, see Section 8.3, "Using the Target Tag to Execute PPR."

Before you begin:

It may be helpful to have an understanding of declarative partial page rendering. For more information, see Section 8.2, "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 more information about naming containers, see Section 4.8, "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.

Example 8–4 shows a `link` component configured to execute PPR.

Example 8–4 Code for Enabling Partial Page Rendering Through a Partial Submit

```
<af:link id="deleteFromCart" partialSubmit="true"
            actionListener="#{homeBean...}" text="Delete From Cart">
```

Example 8–5 shows an `outputText` component that will be rerendered when the link with ID `deleteFromCart` in Example 8–4 is clicked.

Example 8–5 Code for Partial Page Rendering Triggered by Another Component

```
<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. For more information, see Section 8.3, "Using the Target Tag to Execute PPR."

8.2.2 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.

8.2.3 What You May Need to Know About PPR and Screen Readers

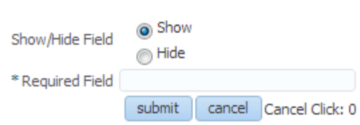
Screen readers do not reread the full page in a partial page request. PPR causes the screen reader to read the page starting from the component that fired the partial page request. You should place the target components after the component that triggers the partial request; otherwise, the screen reader would not read the updated target components.

8.3 Using the Target Tag to Execute PPR

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 Example 8–6.

Example 8–6 Target Tag for Button Determines Which Components to Execute and Render

```
<af:panelFormLayout id="pfl1" 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="pgl9">
    <af:button text="submit" partialSubmit="true" id="cb2"
      <af:target execute="@this it3" render="pfl1"/>
    </af:button>
    <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 Example 8–7.

Example 8–7 Using the Target Tag with LOV Component

```
<af:panelFormLayout id="pfl2">
  <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>
```

8.3.1 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. For more information, see Section 8.3, "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.

Example 8–8 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.

Example 8–8 Using the Target Tag to Avoid Validation Issues

```
<af:panelFormLayout id="pfl1">
  <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="pfl1"/>
    </af:selectBooleanRadio>
    <af:selectBooleanRadio id="hide3" autoSubmit="true" text="Hide" group="group3"
                           value="#{validate.hide3}">
      <af:target execute="hide3 show3" render="pfl1"/>
    </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="pgl9">
    <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="pfl1"/>
    </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>
```

8.4 Enabling Partial Page Rendering Programmatically

When you want a target to be rerendered based on specific logic, you can enable partial page rendering programmatically.

8.4.1 How to Enable Partial Page Rendering Programmatically

You use the addPartialTarget method to enable partial page rendering.

How to enable PPR programatically:

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.

Example 8–9 Rerendering Using Partial Targets

```
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.

8.5 Using Partial Page Navigation

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. For more information, see Section 4.2, "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.

8.5.1 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. For more information, see Section 8.5, "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 rerendered (the header content remains constant for all pages). However, the entire document will be rerendered when an action on the page is defined to use full page submit and also when an action does not result in navigation.

8.5.2 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.

Part III

Creating Your Layout

Part III contains the following chapters:

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:

9.1 About Organizing Content on Web Pages

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. For more information, see Chapter 10, "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, 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. For more information about stretching and other geometry management functionality of layout components, see Section 9.2.1, "Geometry Management and Component Stretching."

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. For more information, see Section 9.2, "Starting to Lay Out a Page."	X	
`form`	Creates an HTML `<form>` element. For more information, see Section 9.2, "Starting to Lay Out a Page."		
Page Layout Containers			
`panelGridLayout`	Used in conjunction with `gridRow` and `gridCell` 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. For more information, see Section 9.3, "Arranging Content in a Grid."	X (when the `gridRow` and `gridCell` components are configured to stretch)	X (when the `dimensionsFrom` attribute is set to `parent`)
`panelStretchLayout`	Contains `top`, `bottom`, `start`, `center`, and `end` facets where you can place other components. For more information, see Section 9.4, "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. For more information, see Section 9.5, "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). For more information, see Section 9.8, "Arranging Contents in a Dashboard."	X	X (when the `dimensionsFrom` attribute is set to `parent`)
`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. For more information, see Section 9.6, "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. For more information, see Section 9.7, "Arranging Content in Forms."		

Table 9–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
Components with Show/Hide Capabilities			
showDetailHeader	Can hide or display contents below the header. Often used as a child to the panelHeader component. For more information, see Section 9.9, "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. For more information, see Section 9.10, "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. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically."	X (if it is being stretched or if the type attribute is set to stretch)	X
panelAccordion	Used in conjunction with showDetailItem components to display as a panel that can be expanded or collapsed. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."		X (when the dimensionsFrom attribute is set to parent)
panelTabbed	Used in conjunction with showDetailItem components to display as a set of tabbed panels. For more information, see Section 9.10, "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 navigationPane component in a navigational menu. For more information, see Section 20.6, "Using Navigation Items for a Page Hierarchy."		X (when the dimensionsFrom attribute is set to parent)
panelDrawer	Used in conjunction with showDetailItem components to display as a set of tabs that can open and close like a drawer. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."		X
panelSpringboard	Used in conjunction with showDetailItem components to display as a set of icons, either in a grid or in a strip. When the user clicks an icon, the associated showDetailItem contents display below the strip. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."		X
showDetail	Hides or displays content through a toggle icon. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically."		
Miscellaneous Containers			

Table 9–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
`panelHeader`	Contains child components and provides a header that can include messages, toolbars, and help topics. For more information, see Section 9.11, "Displaying Items in a Static Box."	X (if the `type` attribute is set to `stretch`)	X (if the `type` attribute is set to `stretch`)
`panelCollection`	Used in conjunction with collection components such as `table`, `tree` and `treeTable` to provide menus, toolbars, and status bars for those components. For more information, see Section 12.9, "Displaying Table Menus, Toolbars, and Status Bars."	X (only a single table, tree, or tree table)	X
`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 `navigationPane` component that is configured to display tabs. For more information, see Section 20.6, "Using Navigation Items for a Page Hierarchy."	X (in the Center facet)	X (when the `dimensionsFrom` attribute is set to `parent`)
`inlineFrame`	Creates an inline `iframe` tag.		X
`navigationPane`	Creates a series of navigation items representing one level in a navigation hierarchy. For more information, see Section 20.6, "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. For more information, see Section 9.12, "Displaying a Bulleted List in One or More Columns."		
`panelWindow`	Displays child components inside a popup window. For more information, see Section 15.2, "Declaratively Creating Popups."		
`toolbox`	Displays child toolbar and menu components together. For more information, see Section 16.3, "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). For more information, see Section 9.13, "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). For more information, see Section 9.13, "Grouping Related Items."		

Table 9–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
Spacing Components			
`separator`	Creates a horizontal line between items. For more information, see Section 9.14, "Separating Content Using Blank Space or Lines."		
`spacer`	Creates an area of blank space. For more information, see Section 9.14, "Separating Content Using Blank Space or Lines."		

9.1.1 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. For more information, see Section 10.3, "Using Page Templates."
- **Themes**: Themes add color styling to some of layout components, such as the `panelBox` component. For more information about themes, see Appendix 31, "Customizing the Appearance Using Styles and Skins"
- **Skins**: You can change the icons and other properties of layout components using skins. for more information, see Chapter 31, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."
- **Accessibility**: You can make your input components accessible. For more information, see Chapter 33, "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. For more information, see Section 3.5.2, "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. For more information, see Chapter 6, "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. For more information, see Chapter 35, "Allowing User Customization on JSF Pages."

9.2 Starting to Lay Out a Page

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. For more information, see Section 9.2.5, "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. For more information, see Section 11.2, "Defining Forms."

JDeveloper automatically inserts the `view`, `document`, and `form` tags for you, as shown in Example 9–1. For more information, see Section 3.4, "Creating a View Page."

Example 9–1 Initial JSF Page Created by JDeveloper Wizard

```
<?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 more information about stretching and flowing, see Chapter 9.2.1, "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. For more information, see Section 9.2.3, "Using Quick Start Layouts."

9.2.1 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.

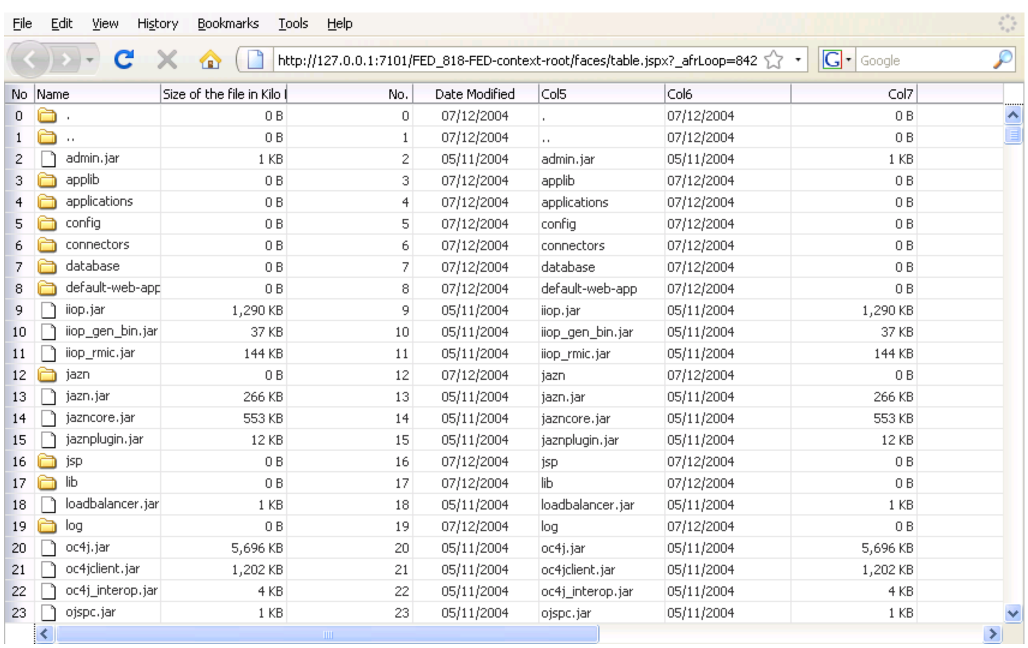
By default, if there is only a single effective visual root component, that root component will stretch automatically to consume the browser's viewable area, provided that component supports geometry management. Examples of geometry management components are `panelGridLayout` and `panelSplitter`. If 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 Chapter 15, "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.

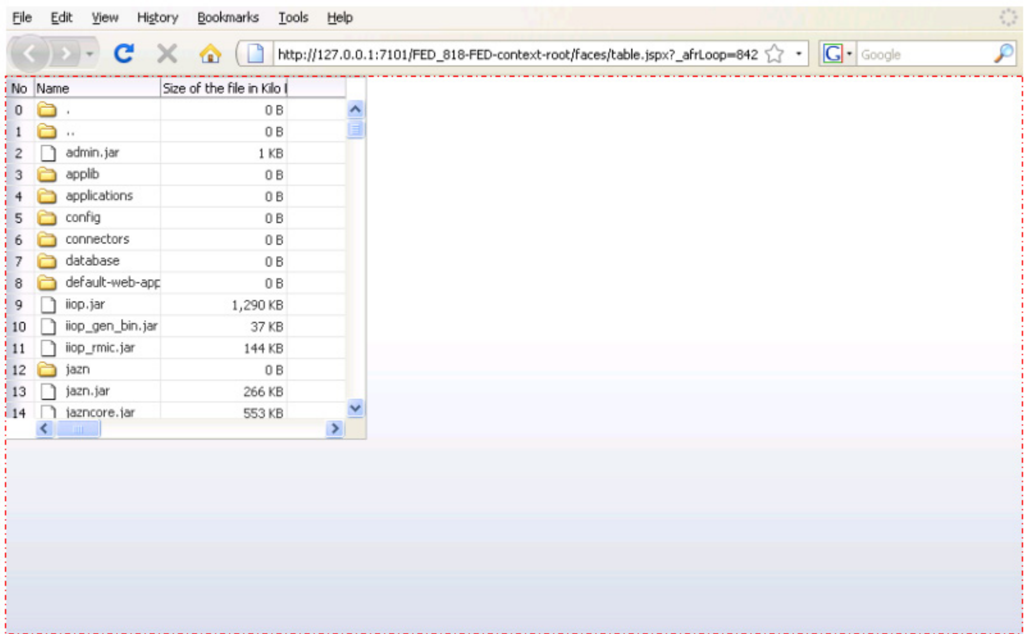
Figure 9–1 Table Inside a Component That Stretches Child Components



No	Name	Size of the file in Kilo I	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.

9.2.2 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)
- `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)

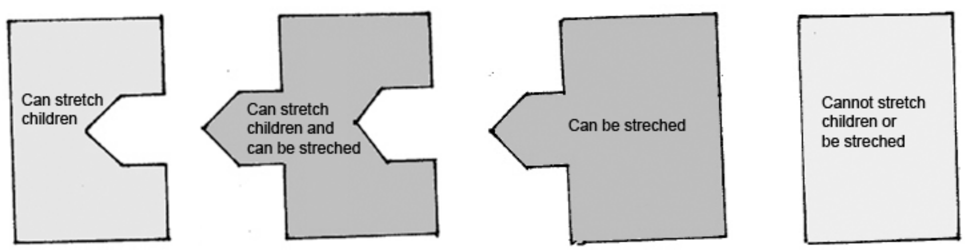
- `region`
- `showDetailHeader` (when configured to stretch)
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

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

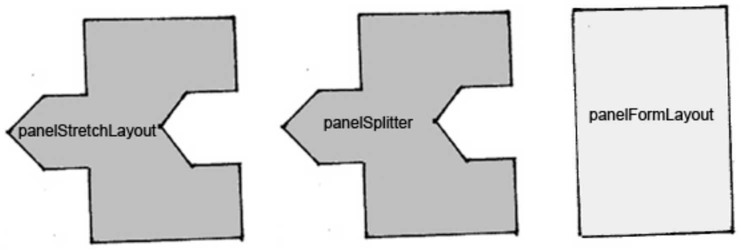
Figure 9–3 Four Categories of Components for Geometry Management



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 `panelForm`Layout 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.

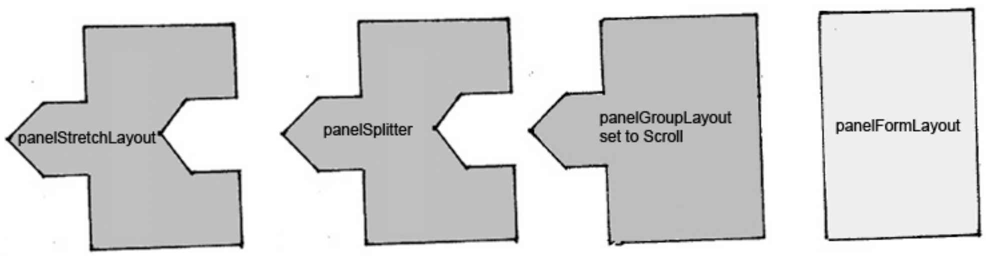
Figure 9–4 Order of Components in One Layout Scenario



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.

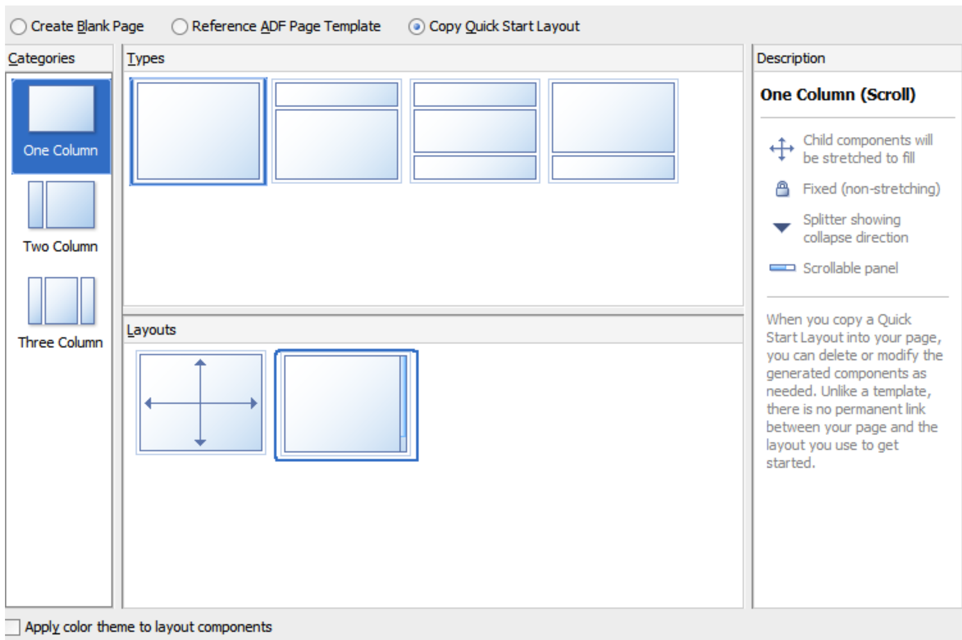
> **Tip:** If you know that you always want your components to stretch or not to stretch based on the parent's settings, then consider setting the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter to `auto`. For more information, see Section A.2.3.29, "Geometry Management for Layout and Table Components."

9.2.3 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 Appendix E, "Quick Start Layout Themes." For more information about themes, see Chapter 31, "Customizing the Appearance Using Styles and Skins"

For more information about creating pages using the quick layouts, see Section 3.4, "Creating a View Page."

9.2.4 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. For more information, see Section 9.2.3, "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. For more information, see Section 9.2.2, "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.

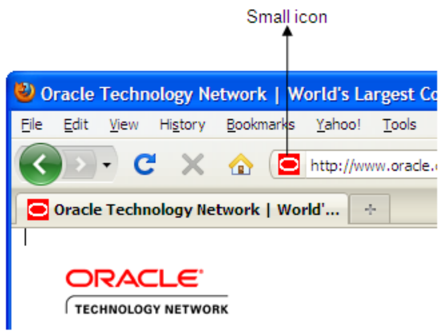
- Never use the `position` style.
- Ensure that the `maximized` attribute on the `document` tag is set to `true` (this is the default). For more information about setting the attribute, see Section 9.2.5, "How to Configure the document Tag."

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.

9.2.5 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. For more information, see Section 4.3, "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. For more information, see Section 9.2.1, "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:

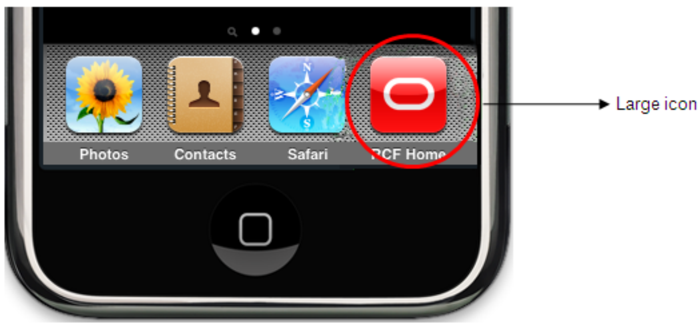
```
/images/small-icon.png /small-icon.ico
```

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.

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 more information about state saving, see Appendix A.2, "Configuration in web.xml."

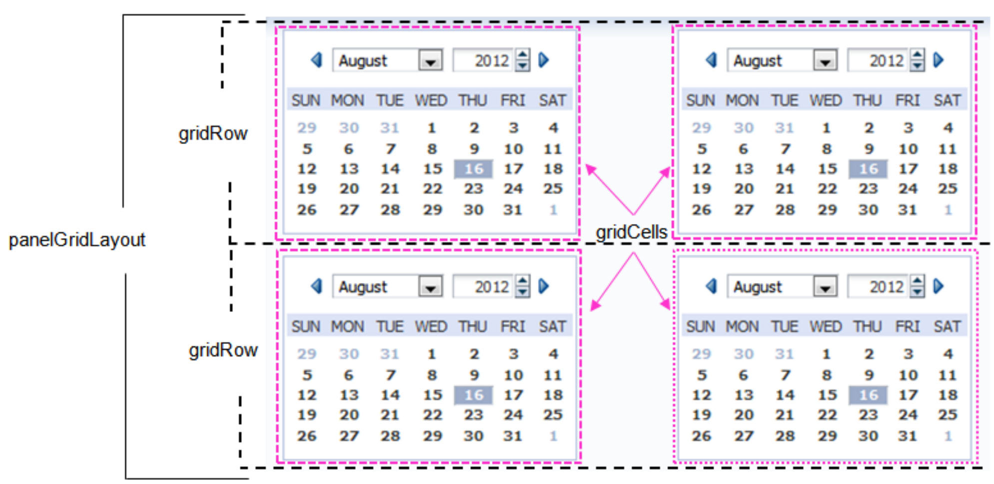
9.3 Arranging Content in a Grid

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.

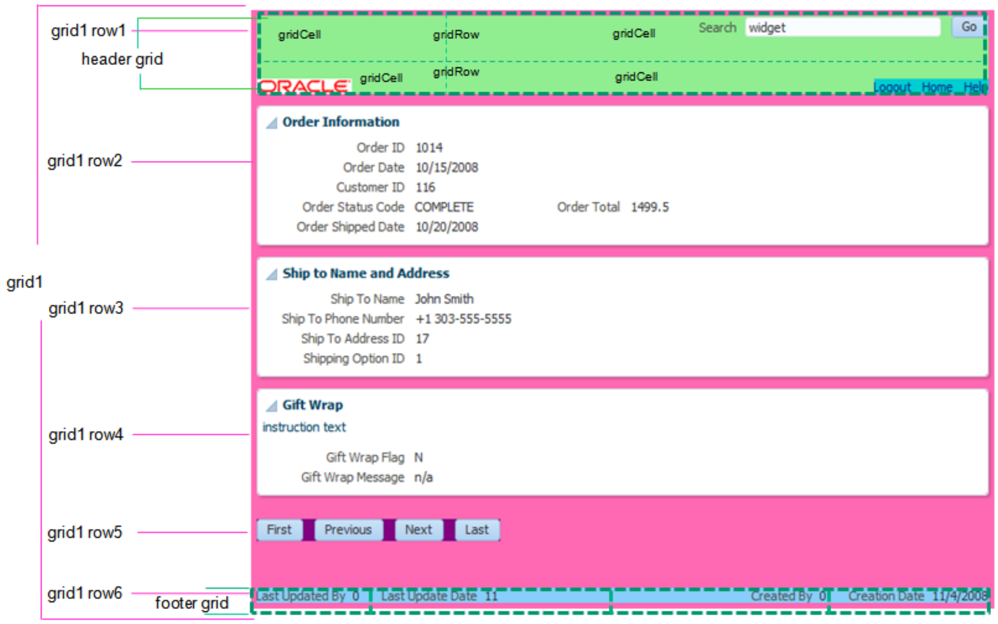
Figure 9–9 shows a `panelGridLayout` component that contains two `gridRow` components. Each of the `gridRow` components contain two `gridCell` components. Each of the `gridCell` components contain one `chooseDate` component.

Figure 9–9 Simple Grid Layout with Two Rows Each with Two Cells



You can nest `panelGridLayout` components. 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 it 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.

If instead 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.

9.3.1 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. For more information, see Section 9.3, "Arranging Content in a Grid."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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. For more information, 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. For more information, see Section A.2.3.29, "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 more 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 more information about getting the `panelGridLayout` component to stretch, see Step 5).

9.3.2 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)
- `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.

9.3.3 What You May Need to Know About Determining the Structure of Your Grid

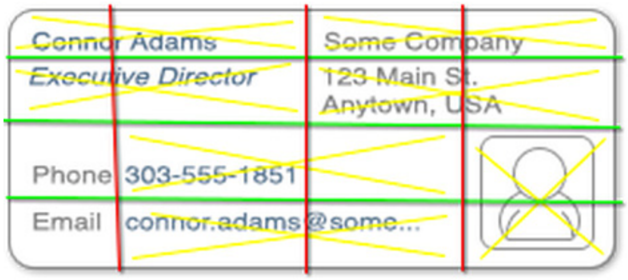
When you are given a mock-up of a page, you may not know how to break it down into a grid. Follow these tips to help determine your columns, rows, and grid separations for consecutive grids.

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.

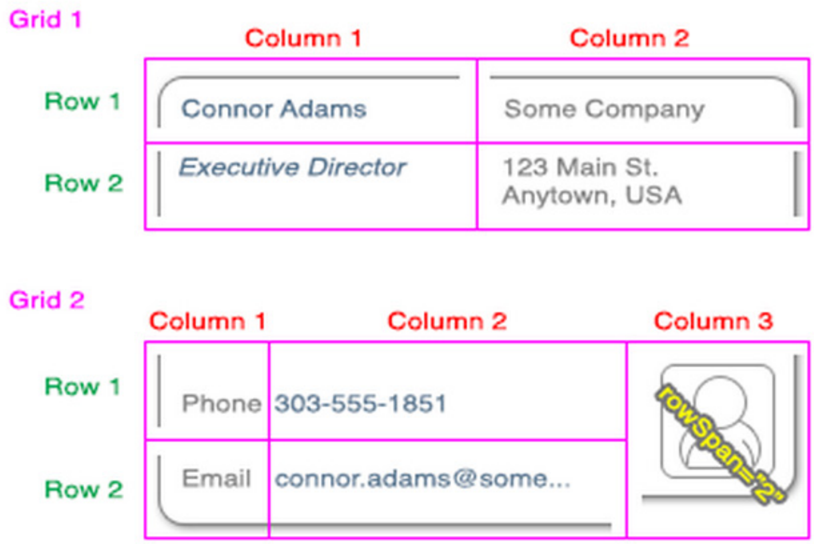
Figure 9–12 Consecutive Grids to Simplify Column Spanning

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:
   >
   > 1. Place the `inputText` components for the phone and email fields in the second column.
   > 2. Set the `simple` attribute on those components to `true`, so the built-in labels don't display.
   > 3. Add new `outputLabel` components to the first column for the labels.

   Figure 9–13 shows the final grid design.

Figure 9–13 Final Grid Design



> **Best Practice Tip:** You should not have more than three layers of `panelGridLayout` components.

9.3.4 What You May Need to Know About Determining Which Layout Component to Use

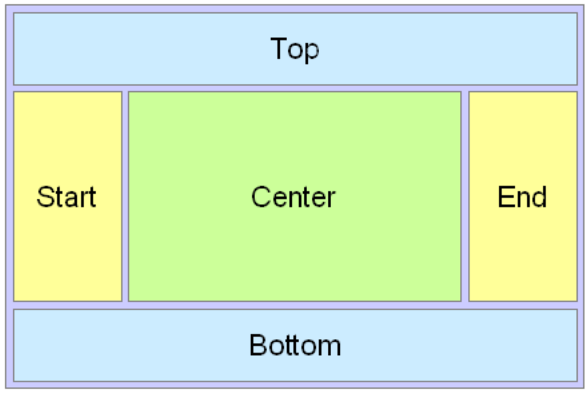
The `panelGridLayout` component provides the most flexibility of the layout components, while producing a fairly small amount of HTML elements. With it, you have full control over how each individual cell is aligned within its boundaries. Conversely, the `panelGroupLayout` provides very little control 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.

9.4 Arranging Contents to Stretch Across a Page

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 9–14 shows the component's facets.

Figure 9–14 Facets in the panelStretchLayout Component



Note: Figure 9–14 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 Section 9.14, "Separating Content Using Blank Space or Lines." Child Components components in the `center` facet are then stretched to fill up any remaining space. For more information about component stretching, see Section 9.2.1, "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. Example 9–2 shows abbreviated code from the `fileExplorerTemplate` file.

Example 9–2 panelStretchLayout in the File Explorer's Template 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.

9.4.1 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. For more information, see Section 9.2.2, "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. For more information, see Section 9.4, "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. For more information, see Section 9.1.1, "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. For more information, see Section A.2.3.29, "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 Section 9.4.2, "What You May Need to Know About Geometry Management and the panelStretchLayout Component," for more details.

   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:

   1. Right-click the `panelStretchLayout` component in the Structure window.
   2. 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.

9.4.2 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)

- 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)

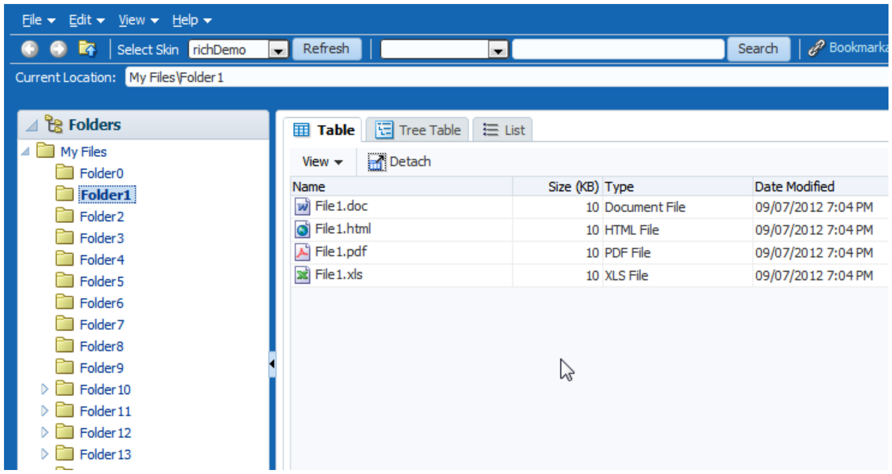
You cannot place components that cannot stretch into facets of a component that stretches its child components. Therefore, if you need to place a component that cannot be stretched into a facet of the `panelStretchLayout` component, wrap that component in a transition component that can stretch.

For example, if you want to place content in a `panelBox` component (which does 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. For more information, see Section 9.2.2, "Nesting Components Inside Components That Allow Stretching."

9.5 Using Splitters to Create Resizable Panes

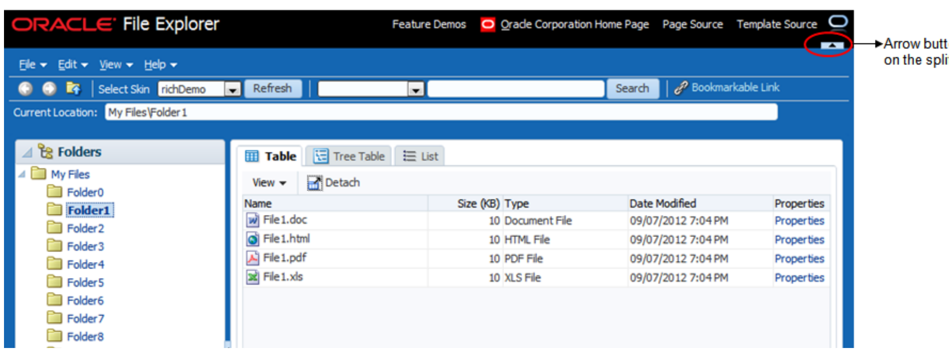
When you have groups of unique content to present to users, consider using the `panelSplitter` component to provide multiple panes separated by adjustable splitters. The ADF Faces demo application uses a `panelSplitter` to separate the component demo area from the editor area, as shown in Figure 9–15. 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 9–15 ADF Faces Demo Application Uses panelSplitter to Separate Contents



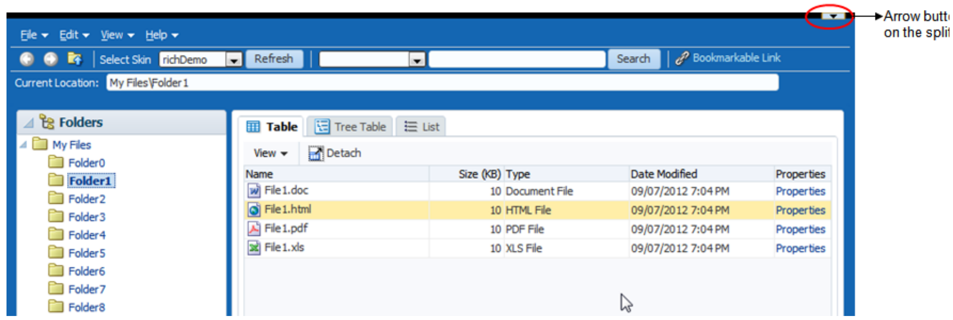
The `panelSplitter` component lets you organize contents into two panes separated by an adjustable splitter. The panes can either line up on a horizontal line (as does the splitter shown in Figure 9–15) or on a vertical line. The ADF Faces demo application uses another `panelSplitter` component to separate the application's global menu from the main body of the page. Figure 9–16 shows the `panelSplitter` component expanded to show the menu, which includes access to the documentation and source.

Figure 9–16 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 9–17.

Figure 9–17 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 Chapter 35, "Allowing User Customization on JSF Pages."

9.5.1 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. For more information, see Section 9.5, "Using Splitters to Create Resizable Panes."

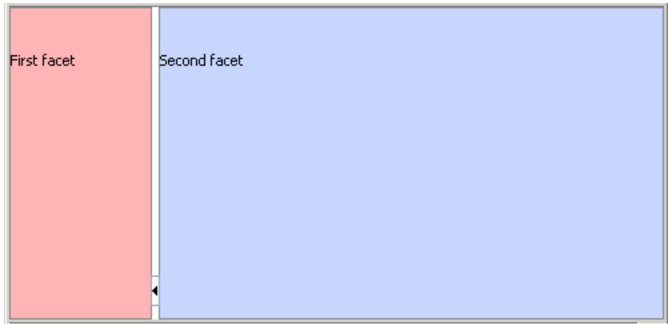
You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "Additional Functionality for Layout Components."

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 9–18.

Figure 9–18 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 9–19 shows the position of the splitter measured 200 pixels from the end panel.

Figure 9–19 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. For more information, see Section A.2.3.29, "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.

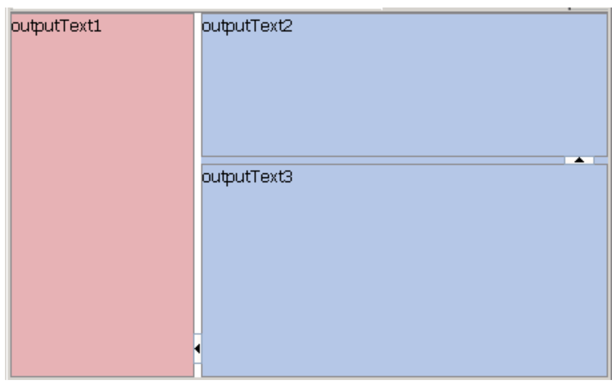
7. To place content in the component, drag and drop the desired component into the `first` and `second` facets. When you have the orientation set to horizontal, the `first` facet is the left facet. When you have the orientation set to vertical, the `first` facet is the top facet. If you want the child component to stretch, it must be a component that supports stretching. For more details, see Section 9.5.2, "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:
>
> 1. Right-click the `panelSplitter` component in the Structure window.
> 2. 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 9–20).

Figure 9–20 Nested panelSplitter Components



Example 9–3 shows the code generated by JDeveloper when you nest splitter components.

Example 9–3 Nested panelSplitter 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 more information about client-side events, see Chapter 6, "Handling Events."

9.5.2 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)
- `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.

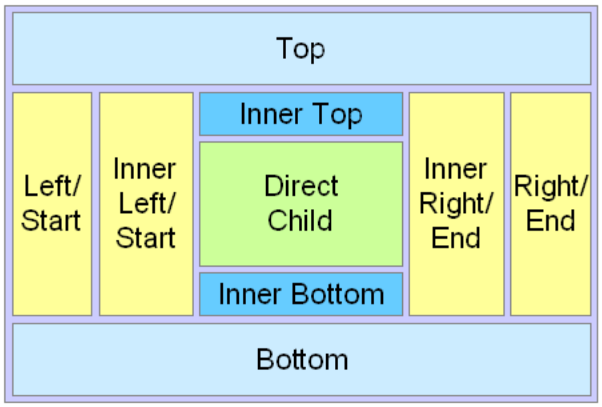
For example, if you want to place content in a `panelBox` component and have it flow within a facet of the `panelSplitter` component, you could place a `panelGroupLayout` component with its layout attribute set to `scroll` in a facet of the `panelSplitter` component, and then place the `panelBox` component in that `panelGroupLayout` component. For more information, see Section 9.2.2, "Nesting Components Inside Components That Allow Stretching."

9.6 Arranging Page Contents in Predefined Fixed Areas

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 9–21 shows the facets of the `panelBorderLayout` component.

Figure 9–21 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. For more information, see Section 9.4, "Arranging Contents to Stretch Across a Page."

9.6.1 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. For more information, see Section 9.6, "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. For more information, see Section 9.1.1, "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. For more information, see

Section 9.13, "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:
   >
   > 1. Right-click the `panelBorderLayout` component in the Structure window.
   > 2. 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.

9.7 Arranging Content in Forms

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 9–22.

Figure 9–22 Right-Aligned Labels and Left-Aligned Fields in a Form

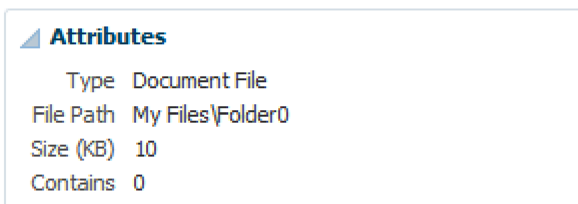
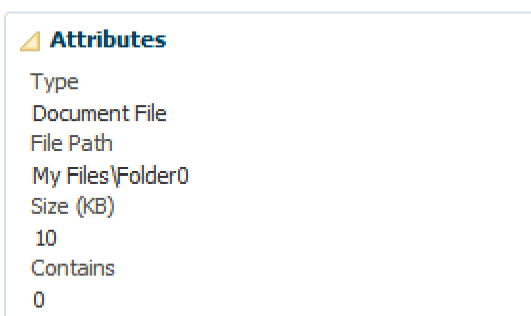


Figure 9–23 shows the same page with the component configured to display the labels above the fields.

Figure 9–23 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. Example 9–4 shows a `panelFormLayout` component with 10 `inputText` child components.

Example 9–4

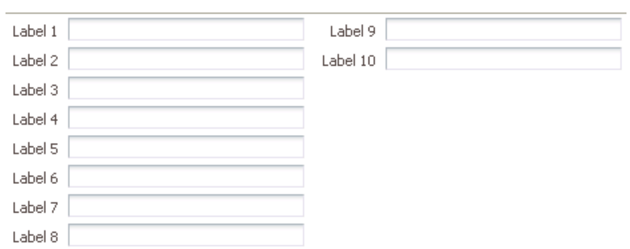
```
<af:panelFormLayout id="pfl1" rows="10">
  <af:inputText label="Label 1" id="it1"/>
  <af:inputText label="Label 2" id="it2"/>
  <af:inputText label="Label 3" id="it3"/>
  <af:inputText label="Label 4" id="it4"/>
  <af:inputText label="Label 5" id="it5"/>
  <af:inputText label="Label 6" id="it6"/>
  <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 9–24.

Figure 9–24 All inputText Components Display in One Column

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 9–25.

Figure 9–25 Components Displayed in Two Columns



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.

9.7.1 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. For more information, see Section 9.7, "Arranging Content in Forms."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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** and **labelWidth** 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.

> **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.

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.

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 `contentStyle` width on an `inputText` component).

---

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.

   Example 9–5 shows the `panelFormLayout` component as it is used on the `properties.jspx` page of the File Explorer application, shown in Figure 9–22.

Example 9–5 panelFormLayout Component

```
<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. For information about using the `panelLabelAndMessage` component, see Section 19.4, "Grouping Components with a Single Label and Message."

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.

   For more information about using the `group` component, see Section 9.7.2, "What You May Need to Know About Using the group Component with the panelFormLayout Component."

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.

   Example 9–6 shows sample code that uses the `panelGroupLayout` component to arrange `footer` child components in a `panelFormLayout` component.

Example 9–6 Footer Child Components in panelFormLayout Arranged Horizontally

```
<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>
```

9.7.2 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 it used to group child components in the `panelFormLayout` component, by default, visible separators can be displayed 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 ever want to have the lines 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.

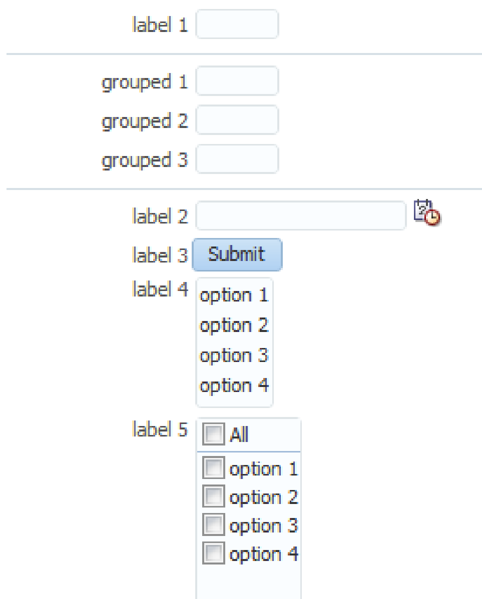
Example 9–7 shows sample code that groups three sets of child components inside a `panelFormLayout` component. The first group is set to hide the separator lines. However, because the second group is configured to display a separator at the start of the group, the lines will display. The second group 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.

Example 9–7 Grouping Child Components in panelFormLayout

```
<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" contentStyle="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" contentStyle="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 9–26.

Figure 9–26 Grouped Components in panelFormLayout



As described in Section 9.7, "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 Example 9–8.

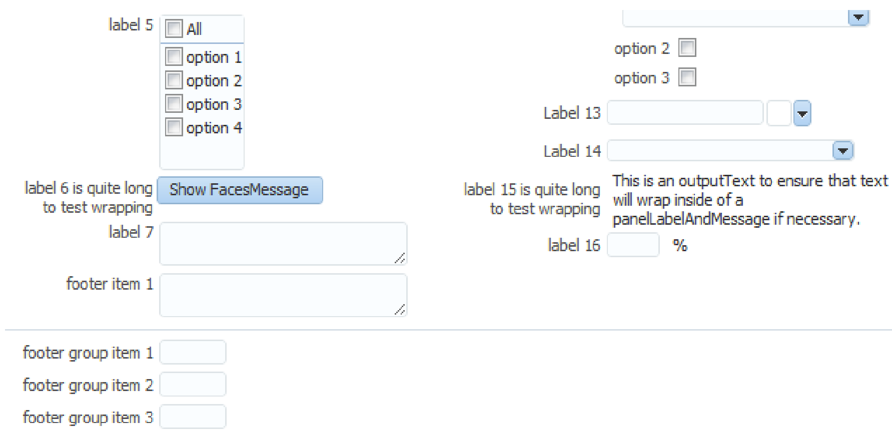
Example 9–8 footer Facet in panelFormLayout with One Root group Component

```
<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: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 9–27.

Figure 9–27 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:outputText value="Another footer item" id="ot6"/>
  </af:group>
</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`.

9.8 Arranging Contents in a Dashboard

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 9–28.

Figure 9–28 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 it 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 9–29 (the color of the dashboard has been changed to fuchsia to make it more easy to see its boundaries).

Figure 9–29 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 9–30.

Figure 9–30 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 9–31, 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 9–31 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. For more information, see Section 36.6, "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.

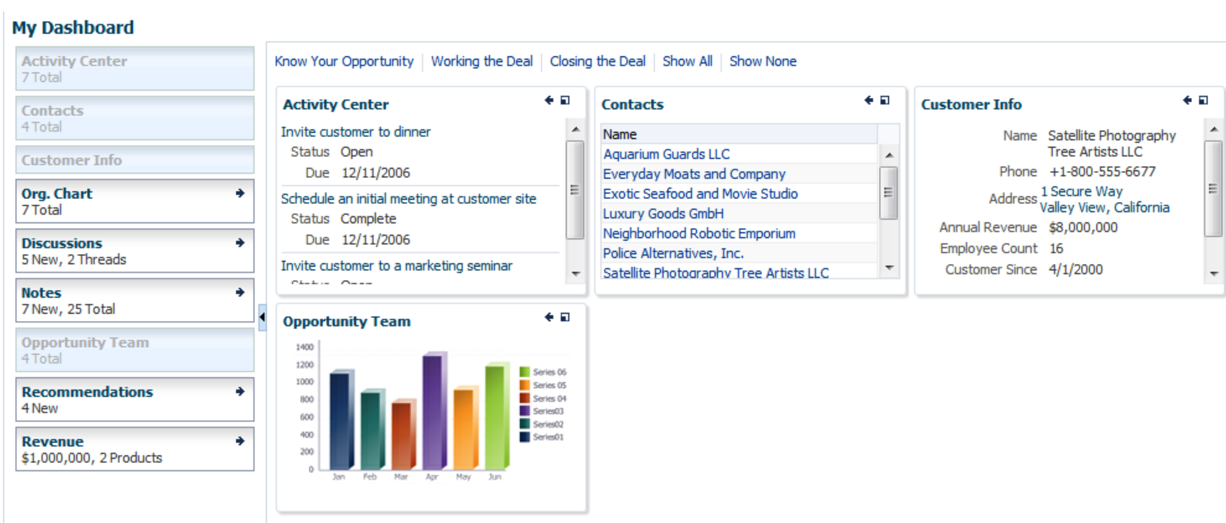
For example, Figure 9–32 shows a `panelDashboard` component used in the right panel of a `panelSplitter` component. In the left panel, list items displayed as links represent each `panelBox` component in the `panelDashboard`. When all `panelBox` components are displayed, the links are all inactive. However, 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 9–32 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 9–33 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 9–33 Practical Example of panelDashboard



9.8.1 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. For more information, see Section 9.8, "Arranging Contents in a Dashboard."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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. For more information, see Section A.2.3.29, "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 more information about creating a handler for a drop event, see Chapter 36, "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. Example 9–9 shows code on a managed bean that handles the insertion of child components.

Example 9–9 Action Listener Code for Insert Button

```
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);

  // Becaue 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.

9.8.2 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.

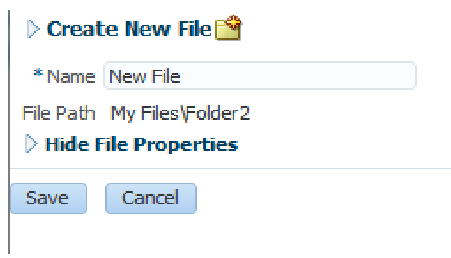
9.9 Displaying and Hiding Contents Dynamically

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.

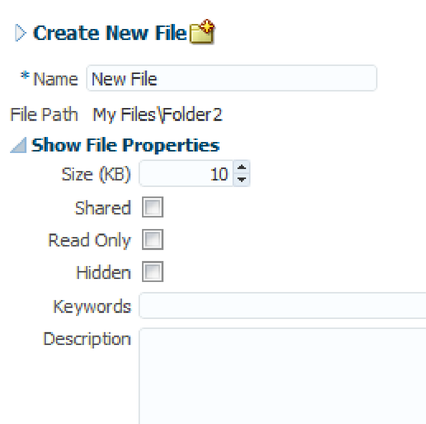
For example, the `newFileItem` page of the File Explorer application uses a `showDetail` component to hide and display file properties. The component is configured to hide the properties when the page is displayed, as shown in Figure 9–34.

Figure 9–34 Collapsed showDetail



When the user clicks the toggle icon, the properties are displayed, as shown in Figure 9–35.

Figure 9–35 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 Section 9.9.1, "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.

> **Tip:** The `showDetailHeader` component is the same as a `panelHeader` component, except that it handles disclosure events. For more information about the `panelHeader` component, see Section 9.11, "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 9–36.

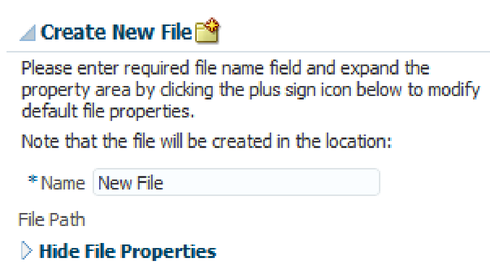
Figure 9–36 Text for the showDetailHeader Is Truncated

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 9–37.

Figure 9–37 Extra Space Is Added Before the Toolbar

Additionally, you can configure the `showDetailHeader` component to be used as a message for errors, warnings, information, or confirmations. The contents 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 9–35. When the user clicks the toggle icon in the header, the contents are disclosed, as shown in Figure 9–38.

Figure 9–38 showDetailHeader Component Used to Display Help

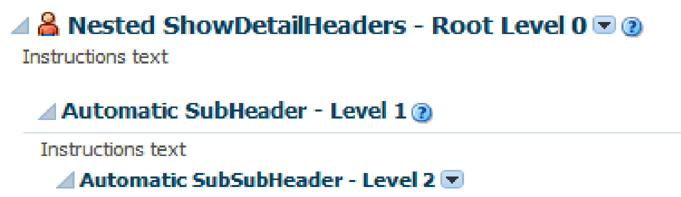


You can also use the `showDetailHeader` component in conjunction with the `panelHeader` component to divide a page into sections and subsections, where some contents can be hidden. The `showDetailHeader` component contains a number of facets, such as a toolbar and menu bar facet. These facets are the same as for the `panelHeader` component. For more information about the `panelHeader` component, see Section 9.11, "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 9–39 shows three nested `showDetailHeader` components, and their different styles.

Figure 9–39 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 9–40.

Figure 9–40 Two panelBox Components

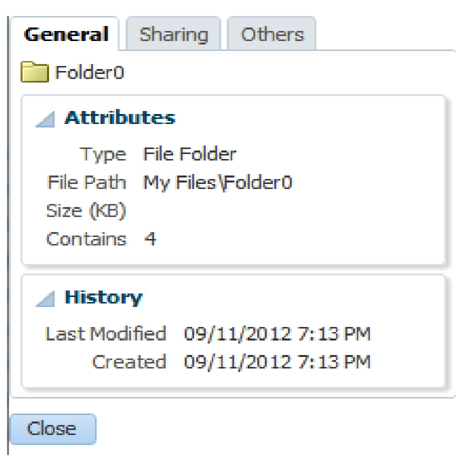
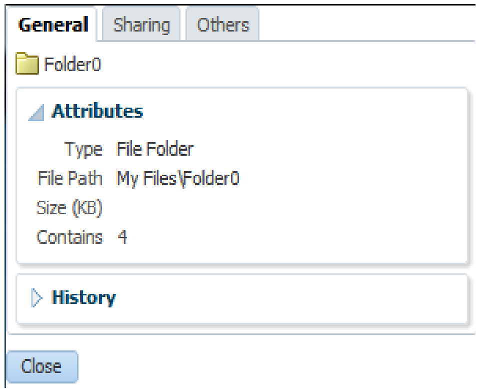


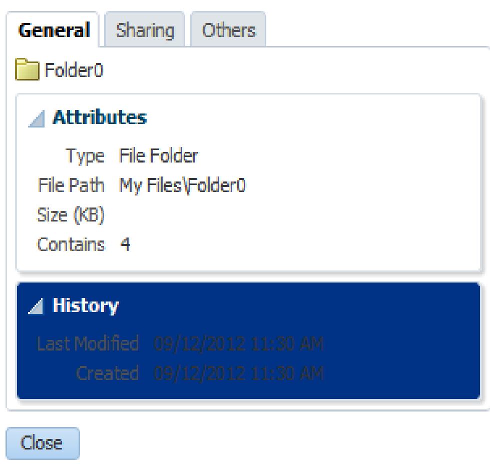
Figure 9–41 shows the same page, but with the History `panelBox` component in an undisclosed state.

Figure 9–41 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 9–42 shows the same panel boxes as in Figure 9–40, but with the bottom `panelBox` component configured to show the medium tone of the core ramp.

Figure 9–42 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.

The `showDetail`, `showDetailHeader`, and `panelBox` components all handle disclosure events on the server. Disclosure events are sent whenever the user toggles the component between disclosed and undisclosed. This means that a roundtrip to the server is required, even though no data may be being sent or received. You can configure these components to so that they handle the disclosure event on the client instead, which improves performance. The event will not be sent to the server until another event is sent, or if the component detects that any data has changed.

If you want to show and hide multiple large areas of content, consider using the `panelAccordion` and `panelTabbed` components. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

9.9.1 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. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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 Chapter 35, "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. For more information, see Section 9.9.5, "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 Section 9.9.4, "What You May Need to Know About Disclosure Events."

4. Set **HandleDisclosure** to `client` if you want the disclosure event to be handled on the client. The event will not be sent to the server until another event is sent, or if the component detects that the data has changed and needs to be updated.

   For information about disclosure events and listeners, see Section 9.9.4, "What You May Need to Know About Disclosure Events."

> **Note:** If you have bound the `disclosureListener` to a listener method that handles the disclosure event, then the `handleDisclosure` value is ignored, and the event is handled on the server.

> **Performance Tip:** If you do not expect the component to handle data changes, you should set the `handleDisclosure` attribute to `client`.

5. To add content, insert the desired child components inside the `showDetail` component.

9.9.2 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. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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.

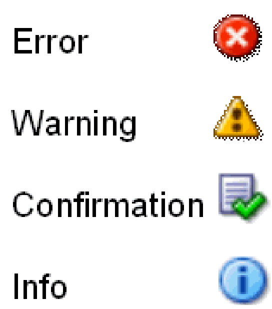
> **Note:** Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

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 9–43 shows each of the icons used for message types.

Figure 9–43 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: 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 Chapter 35, "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 Section 9.9.4, "What You May Need to Know About Disclosure Events."

7. Set **HandleDisclosure** to `client` if you want the disclosure event to be handled on the client. The event will not be sent to the server until another event is sent, or if the component detects that the data has changed and needs to be updated.

   For information about disclosure events and listeners, see Section 9.9.4, "What You May Need to Know About Disclosure Events."

Note: If you have bound the `disclosureListener` to a listener method that handles the disclosure event, then the `handleDisclosure` value is ignored, and the event is handled on the server.

Performance Tip: If you do not expect the component to handle data changes, you should set the `handleDisclosure` attribute to `client`.

8. 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 more information about geometry management, see Section 9.2.1, "Geometry Management and Component Stretching."
9. 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 more information about using the `toolbar` component, see Section 16.3, "Using Toolbars."

   Note: Toolbar overflow is not supported in `panelHeader` components.

10. To add menus to the header, insert menu components into the `menuBar` facet. For more information about creating menus, see Section 16.2, "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.

11. To create a subsection header, insert another `showDetailHeader` component inside an existing `showDetailHeader` component.
12. 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.

---

13. 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 more information, see Section 9.9.6, "What You May Need to Know About Skinning and the showDetailHeader Component."

14. To add content to a section or subsection, insert the desired child components inside the `showDetailHeader` component.

9.9.3 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. For more information, see Section 9.9, "Displaying and Hiding Contents Dynamically."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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. For more information, see Section 9.9.7, "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 **TitleHalign** 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 Section 9.9.4, "What You May Need to Know About Disclosure Events."
8. Set **HandleDisclosure** to `client` if you want the disclosure event to be handled on the client. The event will not be sent to the server until another event is sent, or if the component detects that the data has changed and needs to be updated.

   For information about disclosure events and listeners, see Section 9.9.4, "What You May Need to Know About Disclosure Events."

> **Note:** If you have bound the `disclosureListener` to a listener method that handles the disclosure event, then the `handleDisclosure` value is ignored, and the event is handled on the server.

> **Performance Tip:** If you do not expect the component to handle data changes, you should set the `handleDisclosure` attribute to `client`.

9. 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 Section 16.3, "Using Toolbars."

> **Tip:** If any facet is not visible in the visual editor:
>
> 1. Right-click the `panelBox` component in the Structure window.
> 2. 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.

10. 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.

11. 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. Example 9–10 shows the code you might use for changing the width.

Example 9–10 panelBox Component with inlineStyle Attribute Set

```
<af:panelBox inlineStyle="width:50%;" ...>
  <!-- child contents here -->
</af:panelBox>
```

9.9.4 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, shown in Example 9–11.

Example 9–11 disclosureListener Method

```
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 do not need to use a listener method to react to the disclosure event, then consider setting the `handleDisclosure` attribute to `client`. This setting causes the disclosure event to be handled on the client. The event will not be sent to the server until another event is sent, or if the component detects that the data has changed and needs to be updated. The event will also be automatically sent to the server if the `disclosureListener` attribute is bound to a listener method, even when the `handleDisclosure` attribute is set to `client`.

If you do 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 more information about client-side events and event roots, see Chapter 6, "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. For more information, see Chapter 35, "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.

9.9.5 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:

Example 9–12 Using a Skinning Key to Change the showDetail Indentation

```
af|showDetail {
  -tr-layout: flush;
}
af|showDetail::child-container {
  padding-left: 10px;
}
```

For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

9.9.6 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.

For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

9.9.7 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 pink on the header start cell when the `panelBox` ramp attribute is set to `core` and `background` is set to `default`, you could do the following:

```
af|panelBox::header-start:core:default {background-color:pink; border: none;}
```

You can also use the aliases to change the header and content. For example, `.AFPanelBoxHeaderCoreMedium:alias` is included in the selectors `af|panelBox::header-start:core:medium`, `af|panelBox::header-center:core:medium`, and `af|panelBox::header-end:core:medium`. So if you want to change the background color of the core medium `panelBox` header, you can use the `.AFPanelBoxHeaderCoreMedium:alias` instead of three selectors.

For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

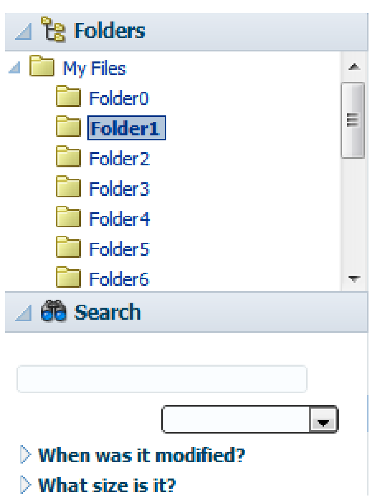
9.10 Displaying or Hiding Contents in Panels

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.

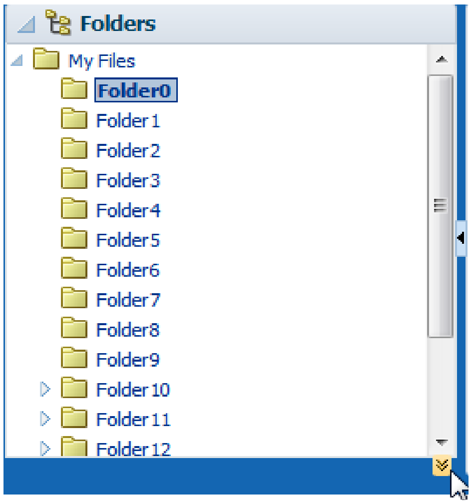
When a panel is collapsed, only the panel header is displayed; when a panel is expanded, the panel contents are displayed beneath the panel header (users can expand the panes by clicking either the `panelAccordion` component's header or the expand icon). The File Explorer application uses the `panelAccordion` component to display the Folders and Search panes, as shown in Figure 9–44.

Figure 9–44 panelAccordion Panes



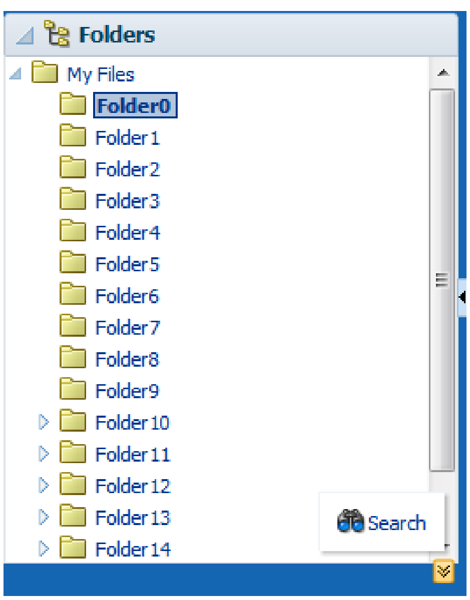
At runtime, when available browser space is less than the space needed to display expanded panel contents, ADF Faces automatically displays overflow icons that enable users to select and navigate to those panes that are out of view. Figure 9–45 shows the overflow icon displayed in the in the lower right-hand corner of the Folders panel of the File Explorer application, when there is not enough room to display the Search panel.

Figure 9–45 Overflow Icon In panelAccordion



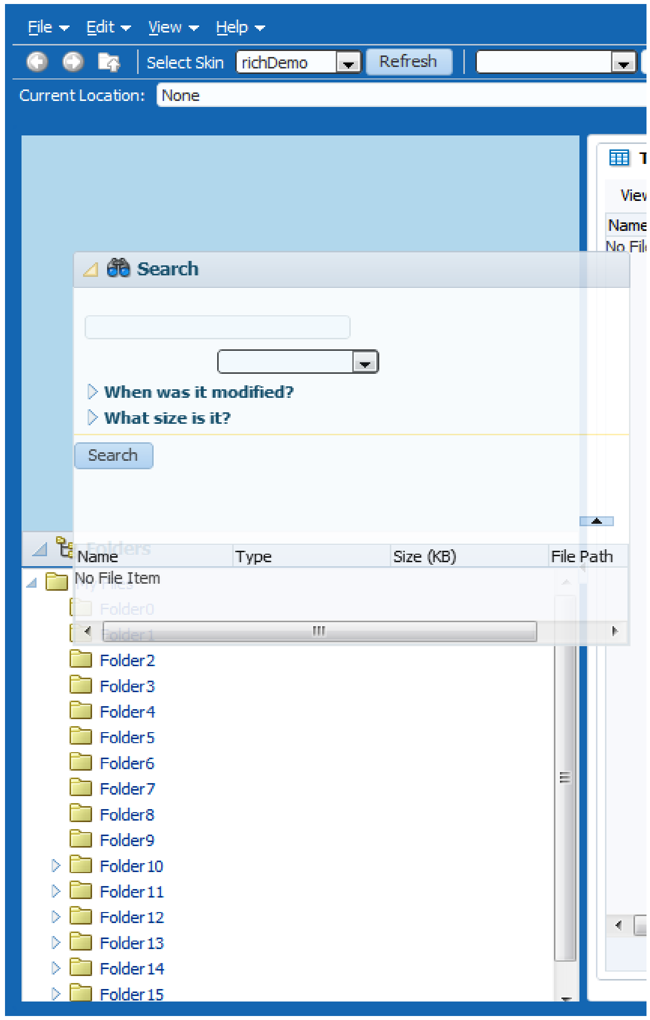
When the user clicks the overflow icon, ADF Faces displays the overflow popup menu (as shown in Figure 9–46) for the user to select and navigate to.

Figure 9–46 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 9–47.

Figure 9–47 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 Section 9.10.1, "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.

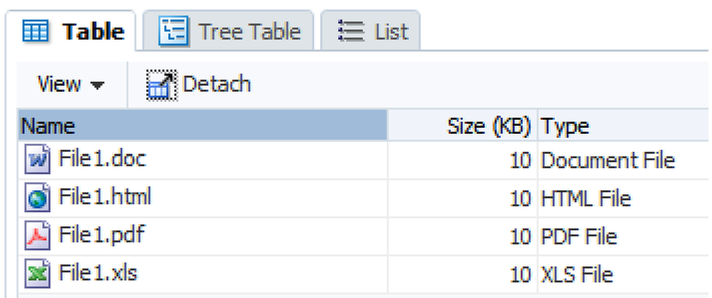
By default, the width of a tab is determined by the text displayed as the label.You can configure the tabs so that instead, the size of the tab is a certain minimum or maximum width. In cases where the text will not fit, you can set an ellipsis to display after the truncated text.

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 9–48.

Figure 9–48 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. For more information, see Section 20.6, "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. For more information, see Section 9.10.9, "What You May Need to Know About Skinning and the panelTabbed Component."

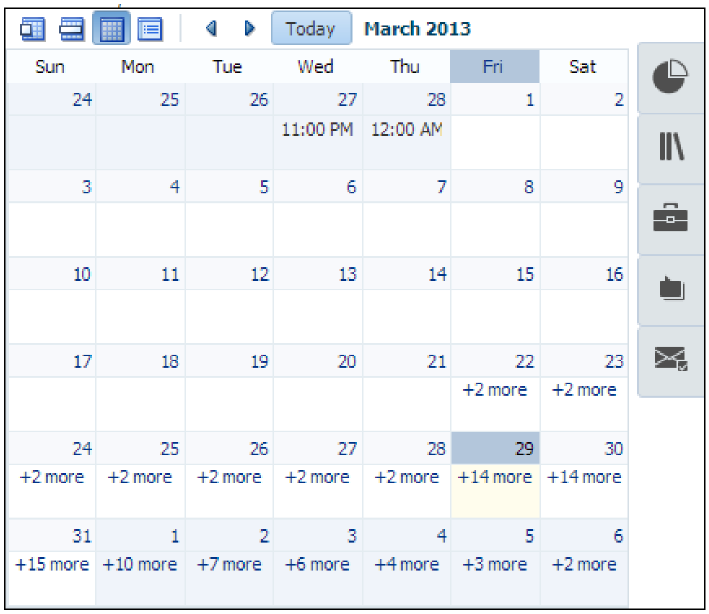
> **Note:** Overflow is only supported when the position attribute is set to `above`, `below`, or `both`.

> **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 Section 9.10.2, "How to Use the panelTabbed Component."

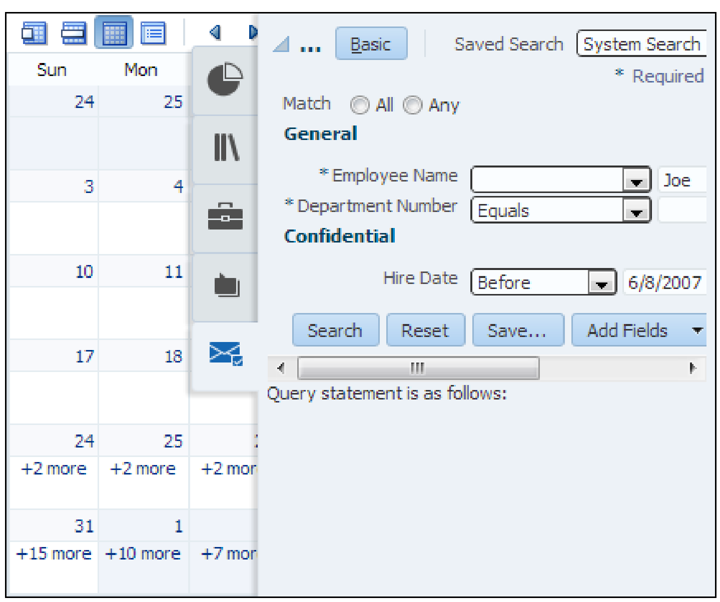
The `panelDrawer` component renders tabs attached to the side of a container component. By default, the drawer aligns to the parent of the `panelDrawer`, but you can choose another close ancestor. It can align to either the start or end of the associated component. When the user clicks a tab, the drawer opens and the content of the child `showDetailItem` becomes visible. Figure 9–49 shows the `panelDrawer` with the drawers closed.

Figure 9–49 panelDrawer Component with Drawers Closed



When the user clicks one of the tabs, the associated drawer opens, as shown in Figure 9–50.

Figure 9–50 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 9–51 shows a `panelSpringboard` component that contains 10 child `showDetailItem` components, configured to display the associated icons in a grid.

Figure 9–51 panelSpringboard Component in Grid Mode

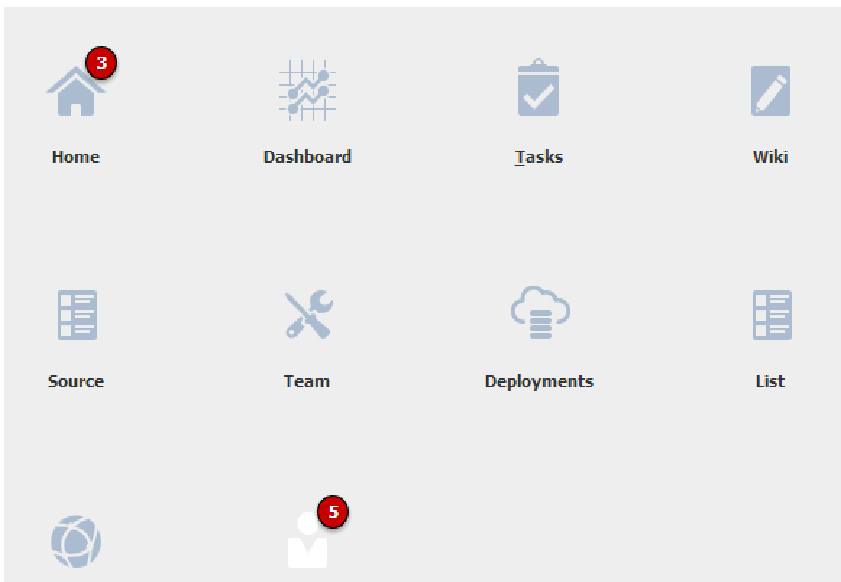
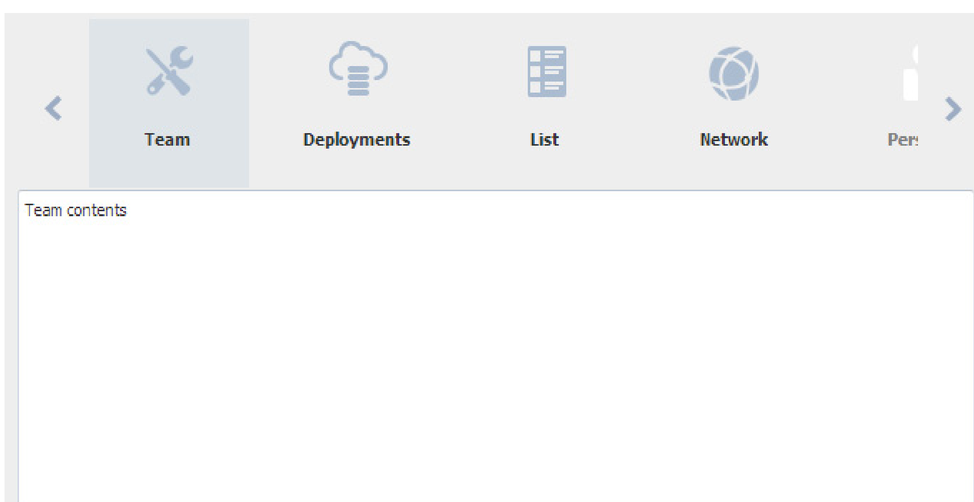


Figure 9–52 shows the same `panelSpringboard` component after clicking the **Team** icon. The `panelSpringboard` icons move to the top, into a strip, and the content associated with the selected icon is displayed.

Figure 9–52 panelSpringboard Component in Strip Mode



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 Section 9.10.6, "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 9–48 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 9–51, badges are used to display a number of items for the Home showDetailItem.

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.

9.10.1 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 Section 9.10.6, "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. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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 Properties window, expand the Common section.
3. Set **DiscloseMany** to true if you want users to be able to expand and see the contents of more than one panel at the same time.

   By default, the value is false. This means only one panel can be expanded at any one time. For example, suppose there is one expanded panel A and one collapsed panel B when the page first loads. If the user expands panel B, panel A will be collapsed, because only one panel can be expanded at any time.
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. For more information, see Section A.2.3.29, "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`: 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 Section 9.10.6, "How to Use the showDetailItem Component to Display Content."

7. By default, all child `showDetailItem` components are created when the `panelTabbed` component is created. If there will be a large number of children, to improve performance you can configure the `panelTabbed` 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 `panelTabbed` component is created.
   - `lazy`: The `showDetailItem` component is created only when the associated tab is selected. Once a tab is selected, the `showDetailItem` component remains created in the component tree.
   - `lazyUncached`: The `showDetailItem` component is created only when the associated tab is selected. Once another tab is selected, the `showDetailItem` component is destroyed.

8. By default, one panel is added for you using a `showDetailItem` component as a child component 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 Section 9.10.6, "How to Use the showDetailItem Component to Display Content."

9.10.2 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. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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 Properties window, expand the Common section.
3. Set **Position** to determine where the tabs should appear. You can set it to one of the following:
   - `above`: The tabs appear above the content. This is the default.
   - `below`: The tabs appear below the content.
   - `start`: The tabs appear on the left, if the reading direction of the client browser is left-to-right (LTR). If the reading direction is right-to-left (RTL), it renders the tabs components on the right.
   - `end`: The tabs appear on the right in LTR locales, and on the left in RTL locales.
   - `left`: The tabs always appear on the left.
   - `right`: The tabs always appear on the right.

   Tip: Setting the `position` attribute to `start` or `end` will allow the position to change based on the reading direction of the locale. For example, setting `position` to `end` means the tabs will appear on the right in the US locale, but on the left in Arabic locales. Setting the `position` to `left` or `right` means the tabs will stay on that side, regardless of the locale.

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).

   For more information, see Section 9.10.6, "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`.

   ---

5. 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.

6. 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. For more information, see Section A.2.3.29, "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: 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. By default, one tabbed panel is created for you using a showDetailItem component as a child to the panelTabbed component. To add more panes, insert the showDetailItem component inside the panelTabbed component. You can add as many tabbed 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 panelTabbed component.

   To add contents for display in a panel, insert the desired child components into each showDetailItem component. For information about using showDetailItem, see Section 9.10.6, "How to Use the showDetailItem Component to Display Content."

9.10.3 How to Use the panelDrawer Component

Using the panelDrawer component to create tabbed panes is similar to using the panelTabbed component to create accordion panes. After adding a panelDrawer component, you insert a series of showDetailItem components to provide the drawer contents for display.

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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 Properties window, expand the Common section.
3. 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.
4. Set **Position** to determine where the tabs for the drawers should appear. If you want the tabs to appear at the end of the aligned component (the right in LTR locale), set it to end (the default). You can also set it to start, left, or right.

> **Tip:** Setting the `position` attribute to `start` or `end` will allow the position to change based on the reading direction of the locale. For example, setting `position` to `end` means the tabs will appear on the right in the US locale, but on the left in Arabic locales. Setting the `position` to `left` or `right` means the tabs will stay on that side, regardless of the locale.

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. 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.
8. Add drawers by inserting `showDetailItem` component insides the `panelDrawer` component. You can add as many drawers as you wish.

   > **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.

   > **Tip:** The `panelDrawer` 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 `panelDrawer` component.

To add contents for display in a drawer, insert the desired child components into each `showDetailItem` component. For information about using `showDetailItem`, see Section 9.10.6, "How to Use the showDetailItem Component to Display Content."

9.10.4 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. For procedures on using the `showDetailItem` component, see Section 9.10.6, "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. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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 9–51. 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 9–52.

   > **Tip:** If you want to be able to switch between the two modes, you need to add Javascript to your page. For more information, see Section 9.10.5, "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 Section 9.10.6, "How to Use the showDetailItem Component to Display Content."

9.10.5 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 Example 9–13.

Example 9–13 Javascript Code to Change DisplayMode to Grid

```
<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);
}
```

You might then call that code from a link, as shown in Example 9–14.

Example 9–14 Page Code to Call Javascript

```
<af:link id="logo" text="Back to Grid">
  <af:clientListener type="click" method="backToGrid"/>
</af:link>
```

For more information about using Javascript on a page, see Chapter 4, "Using ADF Faces Client-Side Architecture."

9.10.6 How to Use the showDetailItem Component to Display Content

Insert showDetailItem components into a panelAccordion, panelTabbed, panelDrawer, or panelSpringboard component only. Each showDetailItem component corresponds to one panel. Typically, you insert two or more showDetailItem components into the parent component. 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. By default, the `disclosed` attribute is `false`, that is, 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 Section 9.10.1, "How to Use the panelAccordion Component," Section 9.10.2, "How to Use the panelTabbed Component," Section 9.10.3, "How to Use the panelDrawer Component,"and Section 9.10.4, "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. For more information, see Section 9.10, "Displaying or Hiding Contents in Panels."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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.
4. To add an icon, set **Icon** to the URI of the image file to use. You can also set **HoverIcon**, **DepressedIcon**, **DisabledIcon**.

   > **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 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 9–46). 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. For more information, see Section 9.10.7, "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 Chapter 35, "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. For more information, see Section 9.10.7, "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 Section 9.9.4, "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.

   By default, the `disclosed` attribute is set to `false`. This means the contents for this panel, tab, or icon are hidden.

   > **Note:** Note the 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 Chapter 35, "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 (supported in the panelAccordion component only), 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. 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 Section 16.3, "Using Toolbars."

    > **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.

9.10.7 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)`
- `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 chid of the `showDetailItem` component, and then place the `panelList` component in that component. For more information, see Section 9.2.1, "Geometry Management and Component Stretching."

9.10.8 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 Section 9.9.4, "What You May Need to Know About Disclosure Events."

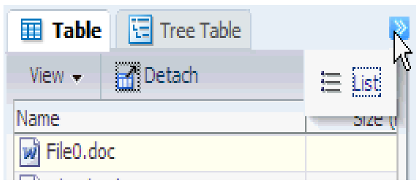
9.10.9 What You May Need to Know About Skinning and the panelTabbed Component

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 compressed layout can display either overflow button(s) or can roll to show hidden tabs, similar to a conveyor belt.

Note: Overflow is only supported when the `position` attribute is set to `above`, `below`, or `both`.

Figure 9–53 shows the overflow compressed layout. When the user clicks the overflow icon a popup displays showing the items that are hidden.

Figure 9–53 Overflow Compressed Layout



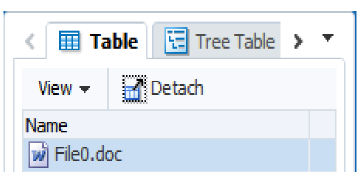
Example 9–15 shows how you use the skinning key to display an overflow layout.

Example 9–15 Using a Skinning Key to Set the Compressed Layout to Overflow

```
af|panelTabbed {
  -tr-layout-type: overflow;
}
```

Figure 9–54 shows the 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 9–54 Conveyor Belt Compressed Layout



Example 9–16 shows how you can use the skinning key to use a conveyor belt layout.

Example 9–16 Using a Skinning Key to Set the Compressed Layout to Conveyor Belt

```
af|panelTabbed {
  -tr-layout-type: conveyor;
}
```

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 more information about skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

9.11 Displaying Items in a Static Box

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.

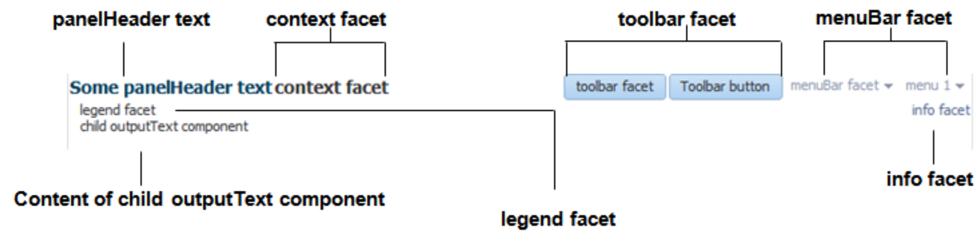
You can use the decorativeBox component when you need to transition to a different look and feel on the page. The decorativeBox component uses themes and skinning keys to control the borders and colors of its different facets. For more information, see Section 9.11.5, "What You May Need to Know About 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.

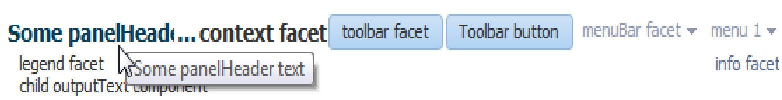
Figure 9–55 shows the different facets in the panelHeader component.

Figure 9–55 panelHeader and Its Facets



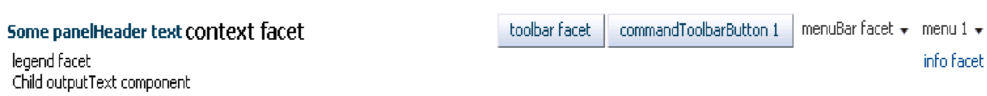
When there is not enough space to display everything in all the facets of the title line, the `panelHeader` text is truncated and displays an ellipsis. When the user hovers over the truncated text, the full text is displayed in a tooltip, as shown in Figure 9–56.

Figure 9–56 Text for the panelHeader Is Truncated



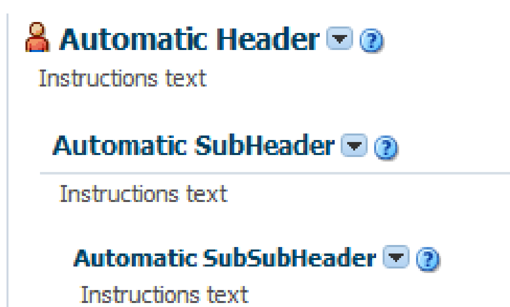
When there is more than enough room to display the contents, the extra space is placed between the `context` facet and the toolbar, as shown in Figure 9–57.

Figure 9–57 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 9–58, you can have a main header with a subheader and then a heading level 1 also with a subheader.

Figure 9–58 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.

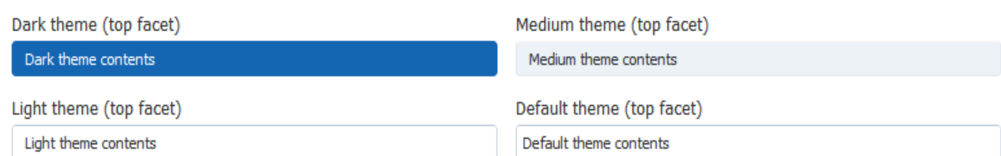
Note: Heading sizes are determined by default by the physical containment of the header components. That is, the first header component will render as a heading level 1. Any header component nested in the first header component will render as a heading level 2, and so on. You can manually override the heading level on individual header components using the `headerLevel` attribute.

For information about using the `panelHeader` component, see Section 9.11.1, "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 9–59 shows the different themes available by default.

Figure 9–59 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 9–60 shows the medium-theme `decorativeBox` configured to stretch to fill its parent, while the dark-theme `decorativeBox` is configured to only be as big as its child `outputText` component.

Figure 9–60 decorativeBox Can Stretch or Not

9.11.1 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. For more information, see Section 9.9.2, "How to Use the showDetailHeader Component."

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 9.11, "Displaying Items in a Static Box."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "Additional Functionality for Layout Components."

To create and use a panelHeader component:

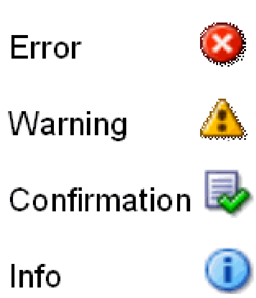
1. In the Components window, from the Layout panel, drag and drop a **Panel Header** onto the page.
2. In the Properties window, expand the **Appearance** section.
3. Set **Text** to the label you want to display for this panel.
4. To add an icon before the label, set **Icon** to the URI of the image 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 9–61 shows the icons used for the different message types.

Figure 9–61 Icons 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.

---

6. To display help for the header, enter the topic ID for **HelpTopicId**. For more information about creating and using help topics, see Section 19.5, "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. For more information, see Section 9.11.4, "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 more information about geometry management, see Section 9.2.1, "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 Section 16.3, "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 Section 16.2, "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:

    1. Right-click the `panelHeader` component in the Structure window.
    2. 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.

9.11.2 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. For more information, see Section 20.6, "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. For more information, see Section A.2.3.29, "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.

For more information, see Section 9.11.3, "What You May Need to Know About Geometry Management and the decorativeBox Component."

9.11.3 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)
- `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. For more information, see Section 9.2.2, "Nesting Components Inside Components That Allow Stretching."

9.11.4 What You May Need to Know About Skinning and the panelHeader Component

Also by default, the style used for heading sizes for the panelHeader 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.

For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

9.11.5 What You May Need to Know About Skinning and the decorativeBox Component

The `decorativeBox` component uses themes and skinning keys to control the borders and colors of its different facets. For example, depending on the skin you are using, 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 boarders 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

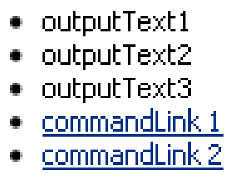
For more information about skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

9.12 Displaying a Bulleted List in One or More Columns

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 9–62. 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 Section 11.6, "Using Selection Components." If you need to create lists that change the model layer, see Chapter 13, "Using List-of-Values Components."

Figure 9–62 PanelList Component with Default Disc Bullet



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.

9.12.1 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. For more information, see Section 9.12, "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. For more information, see Section 9.1.1, "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.

   Example 9–17 shows the code for setting the list style to a circle.

Example 9–17 PanelList Component with ListStyle Attribute Set

```
<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.

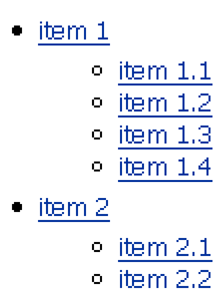
 For example, you could insert a series of `commandLink` components or `outputFormatted` components.

 > **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 Section 9.7, "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.

9.12.2 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 9–63, 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 9–63 Hierarchical List Created Using Nested panelList Components



To achieve the list hierarchy as shown in Figure 9–63, use a `group` component to wrap the components that make up each group of outer items and their respective inner items. Example 9–18 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.

Example 9–18 Nested PanelList Components

```
<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 more information about the `panelGroupLayout` component, see Section 9.13, "Grouping Related Items."

9.13 Grouping Related Items

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.

Example 9–19 shows sample code that groups two sets of child components inside a `panelFormLayout` component.

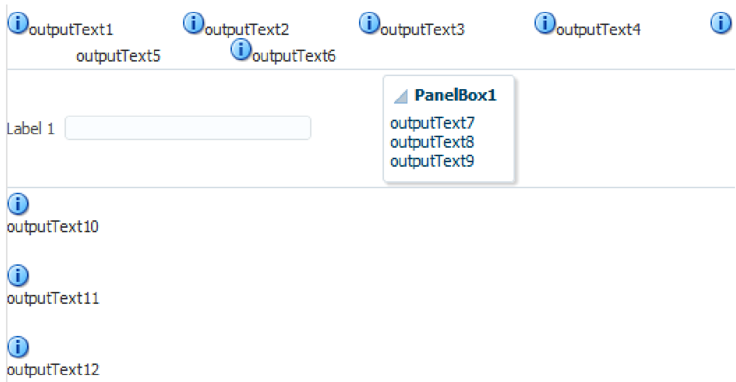
Example 9–19 Grouping Child Components in panelFormLayout

```
<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 9–64. The `layout` attribute value determines the arrangement of the child components.

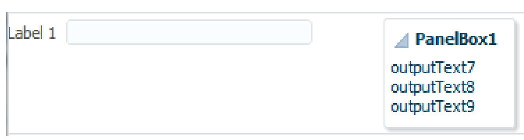
Figure 9–64 panelGroupLayout Arrangements



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. For more information, see Section 9.14, "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 9–65.

Figure 9–65 Top-Aligned Horizontal Layout with panelGroupLayout



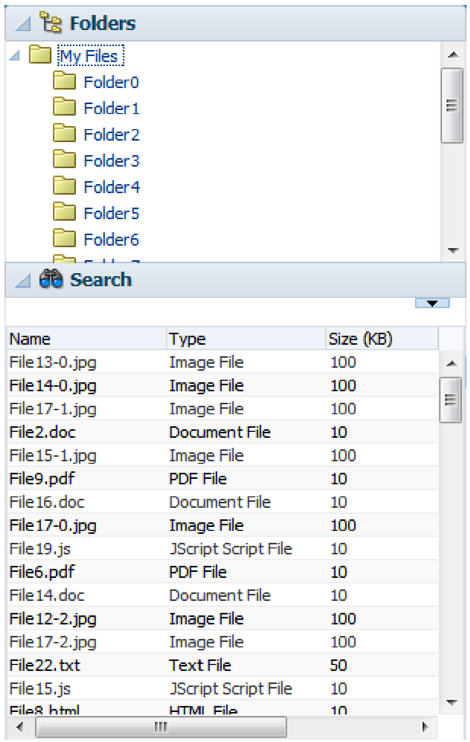
Unlike the `panelSplitter` or `panelStretchLayout` components, the `panelGroupLayout` component does not stretch its child components. Suppose you are already using a `panelSplitter` or `panelStretchLayout` component as the root component for the page, and you have a large number of child components 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 the 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 9–66.

Figure 9–66 Scrollbars Rendered Using panelGroupLayout



9.13.1 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. For more information, see Section 9.13, "Grouping Related Items."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 9.1.1, "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:

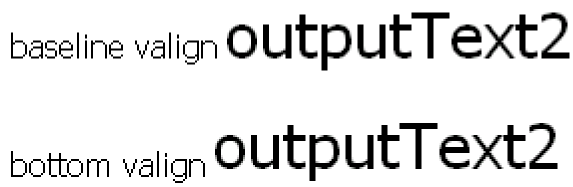
     – **halign**: Sets the horizontal alignment. The default is `center`. Other acceptable values are: `start`, `end`, `left`, `right`.

       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 9–67. 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 9–67 Bottom and Baseline Vertical Alignment of Text



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.

9.13.2 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`.

9.14 Separating Content Using Blank Space or Lines

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.

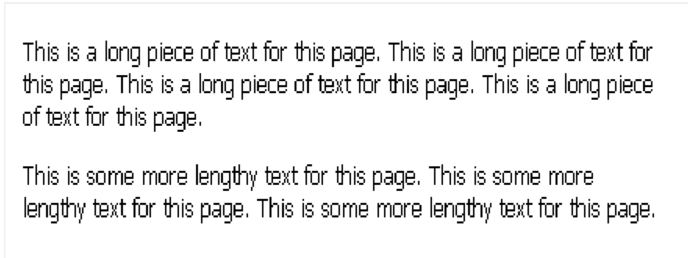
The `height` attribute determines the amount of vertical space to include in the page. Example 9–20 shows a page set up to space out two lengthy `outputText` components with some vertical space.

Example 9–20 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 9–68 shows the effect the `spacer` component has on the page output as viewed in a browser.

Figure 9–68 Vertical Space Viewed in a Browser



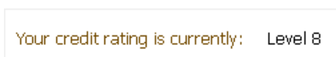
The `width` attribute determines the amount of horizontal space to include between components. Example 9–21 shows part of the source of a page set up to space out two components horizontally.

Example 9–21 Horizontal Space

```
<af:outputLabel value="Your credit rating is currently:"/>
<af:spacer width="10"/>
<af:outputText value="Level 8"/>
```

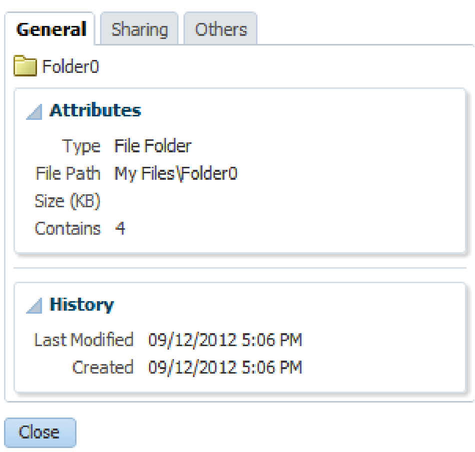
Figure 9–69 shows the effect of spacing components horizontally as viewed in a browser.

Figure 9–69 Horizontal Space Viewed in a Browser



The `separator` component creates a horizontal line. Figure 9–70 shows the `properties.jspx` file as it would be displayed with a `separator` component inserted between the two `panelBox` components.

Figure 9–70 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.

9.14.1 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. For more information, see Section 9.14, "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. For more information, see Section 9.1.1, "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.

9.14.2 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. For more information, see Section 9.14, "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. For more information, see Section 9.1.1, "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:

10.1 About Reusable Content

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 details about creating and using page templates, see Section 10.3, "Using Page Templates," and Section 10.3.3, "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 details about creating and using page fragments, see Section 10.4, "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 details about creating and using declarative components, see Section 10.5, "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 the "Using Task Flows as Regions" chapter of *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 more information about naming containers, see Section 4.8, "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. For more information, see Section 10.6, "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.

10.1.1 Reusable Components Use Cases and Examples

The File Explorer application uses a `fileExploreorTemplate` 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.

10.1.2 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 more information about customization, see the "Customizing Applications with MDS" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.
- For more information on using the Quick Start Layouts to provide a preconfigured layout, see Section 9.2.3, "Using Quick Start Layouts."
- For information about using model parameters and ADF Model data bindings, see the "Using Page Templates" section 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 the "Reusing Application Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

10.2 Common Functionality in Reusable Components

Page templates and declarative components share several common functionalities.

10.2.1 Page in Request Scope

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 more information about scopes, see Section 5.6, "Object Scope Lifecycles."

10.2.2 Access to Child Components for Customization

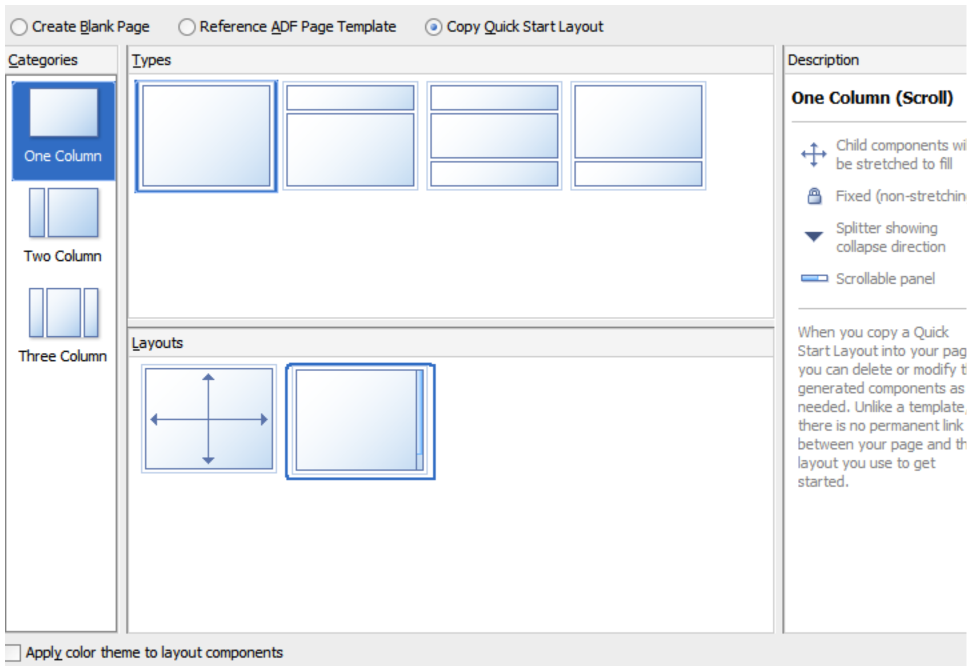
You can control whether child components of a page template or declarative component can be changed by external reference. For example, you can enable or disable the customization of the child components. Both `af:pageTemplateDef` and `af:componentDef` has a `definition` attribute that controls access. 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.

For more information about customization, see the "Customizing Applications with MDS" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

10.3 Using Page Templates

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 more information about the layout components, see Chapter 9, "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 Example 10–1.

Example 10–1 Facet Definition in a Template

```
<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 Example 10–2, 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.

Example 10–2 Facet References in a Page Template

```
<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. Example 10–2 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, as shown in Example 10–3

Example 10–3 Using Page Templates Facets in a JSF Page

```
<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="navigatorsSize" 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 Example 10–4.

Example 10–4 Page Template AttributeDefinition

```
<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 Section 32.3, "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. For more information, see Section 10.6, "Adding Resources to Pages."

10.3.1 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. For more information, see Section 10.3, "Using Page Templates."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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 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. 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 the "Using Page Templates" section 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. Example 10–5 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.

Example 10–5 Component Metadata in Page Template Definition

```
<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. For more information, see Chapter 9, "Organizing Content on Web Pages."

   Declarative components and databound components may be used in the layout section. For information about using declarative components, see Section 10.5, "Using Declarative Components." For information about using databound components in page templates, see the "Using Page Templates" section 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 more information about ADF Libraries, see the "Reusing Application Components" chapter of *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 Example 10–5, 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 more information about using fragments, see Section 10.4.3, "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.

10.3.2 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, the "Using Page Templates" section 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. Example 10–6 shows an example of the `pagetemplate-metadata.xml` file.

Example 10–6 Sample 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.

10.3.3 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 the "Reusing Application Components" chapter of *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. For more information, see Section 10.3, "Using Page Templates."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "Additional Functionality for Reusable Components."

To create a JSF page or page fragment based on a page template:

1. Follow the instructions in Section 3.4.1, "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 more information about using ADF Model data binding, see *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 (see Example 10–7).

   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.

5. To include resources, such as CSS or JavaScript, you need to use the `af:resource` tag. For more information, see Section 10.6, "Adding Resources to Pages."

10.3.4 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 Example 10–7. 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.

Example 10–7 JSF Page that References a Page Template

```
<?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>
```

10.3.5 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 the "Using Page Templates" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

10.3.6 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. For more details, see Section 6.3.8, "What You May Need to Know About Using Naming Containers."

10.4 Using 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 more information about scopes, see Section 5.6, "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.

Example 10–8 shows the abbreviated code for the included `header.jspx` page fragment. Note that it does not contain an `f:view` or `document` tag.

Example 10–8 header.jspx Page Fragment

```
<?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>
```

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 Example 10–9, which is abbreviated code from the `index.jspx` page.

Example 10–9 File Explorer Index JSF Page Includes Fragments

```
<?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 the "Getting Started with ADF Task Flows" chapter of *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 Example 10–10. For information about creating and applying page templates, see Section 10.3, "Using Page Templates," and Section 10.3.3, "How to Create JSF Pages Based on Page Templates."

Example 10–10 Page Fragment Based on a Template

```
<?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>
```

10.4.1 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. For more information, see Section 10.4, "Using Page Fragments."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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 more information about using page templates, see Section 10.3.3, "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. For more information, see Section 9.2.3, "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`.

Example 10–11 shows an example of a page fragment that contains a menu component.

Example 10–11 Page Fragment Sample

```
<?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>
```

10.4.2 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. Example 10–12 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.

Example 10–12 Entries in web.xml for Recognizing and Interpreting .jsff Files

```
<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.

10.4.3 How to Use a Page Fragment in a JSF Page

To consume a page fragment in a JSF page, add the page using either the Components window or the Applications window.

10.4.3.1 Adding a Page Fragment 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. For more information, see Section 10.4, "Using Page Fragments."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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.

10.4.3.2 Adding a Page Fragment Using the Applications Window

You can drag and drop the page fragment directly onto the page.

Before you begin:

It may be helpful to have an understanding of page fragments. For more information, see Section 10.4, "Using Page Fragments."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "Additional Functionality for Reusable Components."

To add a page fragment using the Applications Window:

1. In the Applications window, drag and drop the page fragment onto the page.
2. In the Confirm Add Subview Element dialog, click **Yes**.

10.4.4 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.

10.5 Using Declarative Components

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. For more information, see Section 10.3, "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. For more information, see Section 10.5.1, "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.

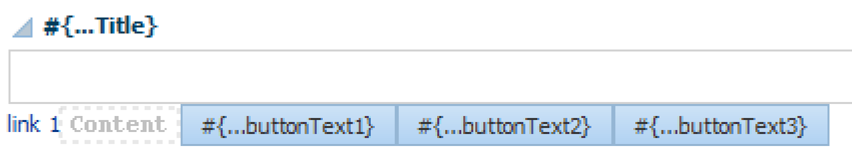
For example, say you want to create a declarative component that uses a `panelBox` component. In the `panelBox` component's toolbar, you want to include three buttons that can be used to invoke `actionEvent` methods on a backing bean. To do this, create the following:

- 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.

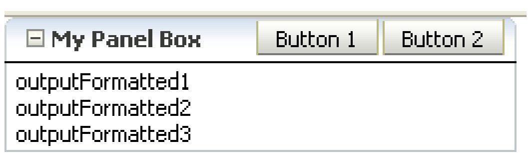
Figure 10–2 Declarative Component in the Visual Editor



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. For more information, see Section 10.6, "Adding Resources to Pages."

10.5.1 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. For more information, see Section 10.5, "Using Declarative Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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. Possible class types to use are: `java.lang.String`, `int`, `boolean`, and `float`. 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 more information about using the ADF Model layer, see the "Using ADF Model in a Fusion Web Application" chapter of *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.

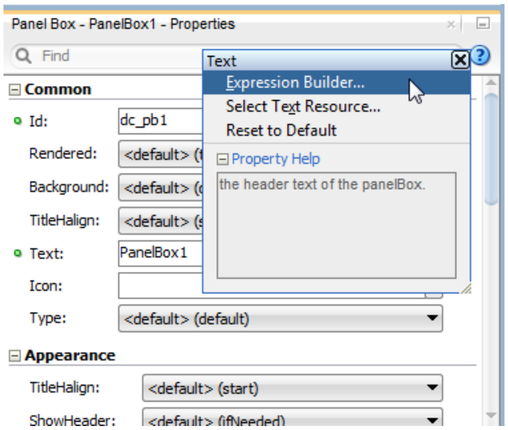
    When you drag **Facet** from the Components window Core Structure panel and drop it in the desired location on the page, JDeveloper displays the Insert Facet Definition dialog. In that dialog, select a facet name from the dropdown list, or enter a facet name, and click **OK**. If you enter a facet name 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.

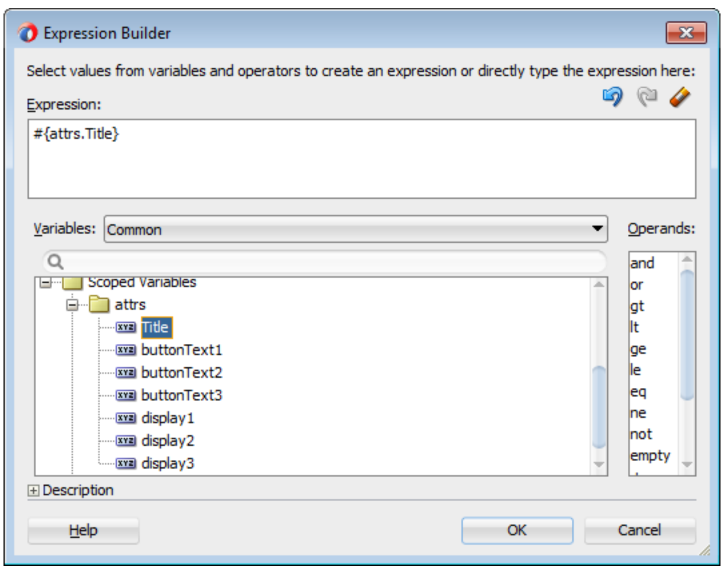
    For example, if you have defined a `Title` attribute and added a `panelBox` as a component, you might use the dropdown menu next to the text attribute in the Properties window to open the Expression Builder, as shown in Figure 10–4.

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



14. 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**.

10.5.2 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. Example 10–13 shows abbreviated code for the declarative component shown in Figure 10–2.

Example 10–13 Declarative Component Metadata in the xmlContent Tag

```
<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. Example 10–14 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.

Example 10–14 Components in a Declarative 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}"
                                 rendered="#{attrs.display2}"
                                 actionListener="#{component.handleMethod2}"/>
        <af:button text="#{attrs.buttonText3}"
                                 rendered="#{attrs.display3}"
                                 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. Example 10–15 shows an example of the `declarativecomp-metadata.xml` file.

Example 10–15 Sample 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.

10.5.3 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 the "Reusing Application Components" chapter of *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:

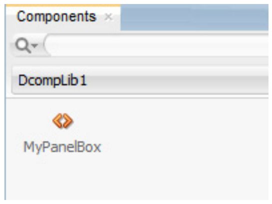
- A component tag class (for example, the `componentDef1Tag.class`) for each declarative component definition (that is, for each `componentDef` component)
- One or more JSP TLD files for the declarative components, using information from the project's `declarativecomp-metadata.xml` file

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 the "Reusing Application Components" chapter of *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.

10.5.4 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 the "Reusing Application Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of declarative components. For more information, see Section 10.5, "Using Declarative Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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. For more information, see Section 10.6, "Adding Resources to Pages."

10.5.5 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. Example 10–16 shows the code for the `MyPanelBox` declarative component to which a user has added a `panelGroupLayout` component that contains three `outputFormatted` components.

Example 10–16 JSF Code for a Declarative Component that Contains Content

```
<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>
```

10.5.6 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.

10.6 Adding Resources to Pages

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 Chapter 31, "Customizing the Appearance Using Styles and Skins."

10.6.1 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. For more information, see Section 10.6, "Adding Resources to Pages."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 10.1.2, "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**.

10.6.2 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 contains the following chapters:

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`).

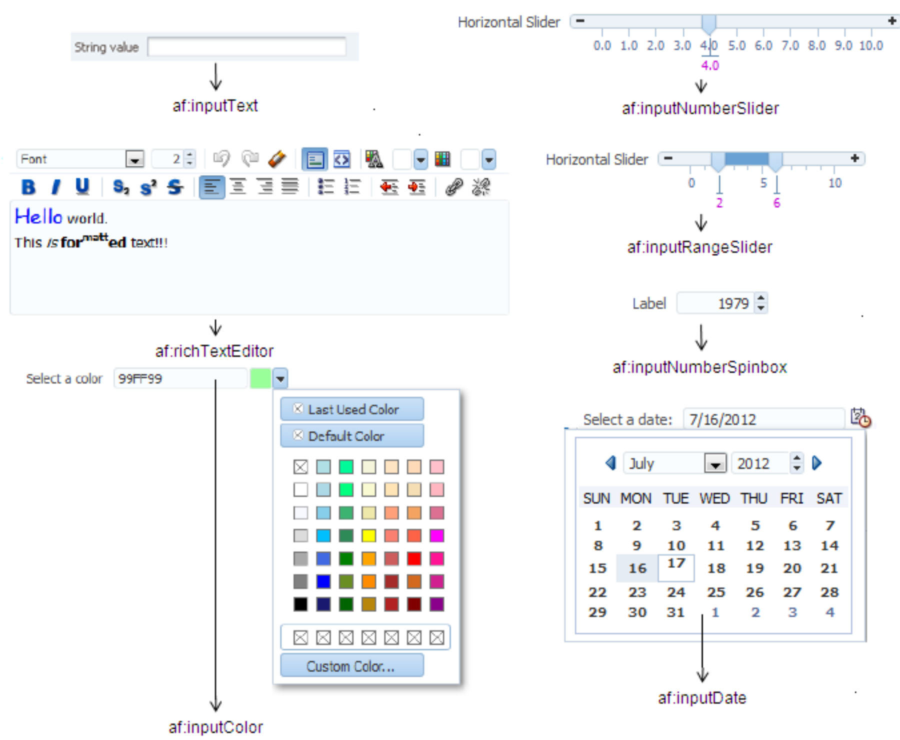
This chapter includes the following sections:

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 more information about those components, see Chapter 13, "Using List-of-Values Components."

11.1 About Input Components and Forms

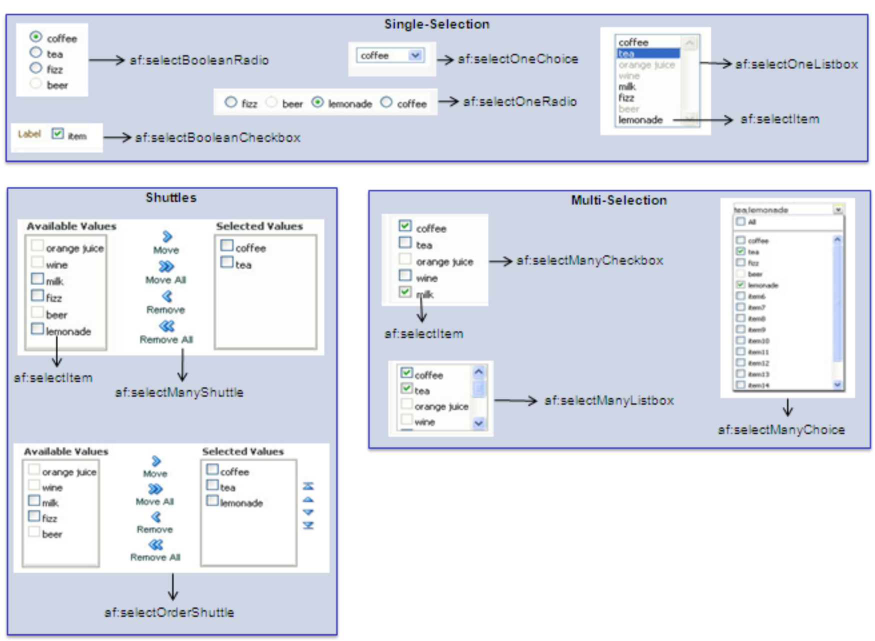
Input components accept user input in a variety of formats. The most common formats are text, numbers, date, and selection lists that appear inside a form and are submitted when the form is submitted. The entered values or selections may be validated and converted before they are processed further. Figure 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.

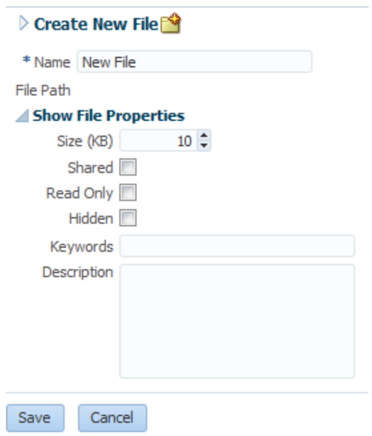
Figure 11–2 Select Components



11.1.1 Input Component Use Cases and Examples

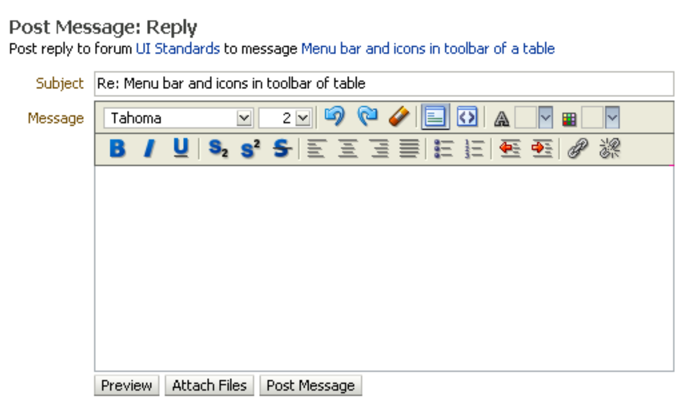
Input components are often used to build forms for user input. For example, the File Explorer application contains a form that allows users to create a new file. As shown in Figure 11–3, input components allow users to enter the name, the size, select permissions, and add keywords, and a description 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



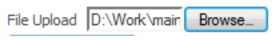
The `richTextEditor` component provides rich text input that can span many lines and can be formatted using different fonts, sizes, justification, and other editing features that may be required when you want users to enter more than simple text. For example, the `richTextEditor` might be used in a web-based 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



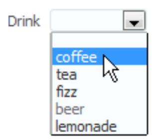
The `inputFile` component allows users to browse for a local file to upload to the application server. For example, an email message might allow users to attach a file to a message, as shown in Figure 11–5.

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.

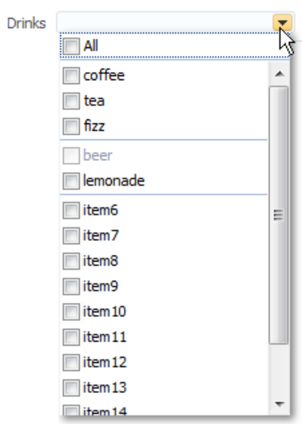
Figure 11–6 Users Can Select One Value From the selectOneChoice Component



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 more information about using LOV components, see Chapter 13, "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.

11.1.2 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. For more information, see Section 3.5.2, "How to Use the EL Format Tags."
- **Client components**: Input components can be client components. To work with the components on the client, see Chapter 4, "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. For more information, 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. For more information, see Chapter 6, "Handling Events."
- You can add validation and conversion to input components. For more information, see Chapter 7, "Validating and Converting Input."
- You can display tips and messages, as well as associate online help with input components. For more information, see Chapter 19, "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. For more information, see Section 5.2, "Using the Immediate Attribute."
- You may want other components on the page to update based on selections you make from a selection component. For more information, see Section 5.3, "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. For more information, see Section 6.6.1, "How to Use the scrollComponentIntoViewBehavior Tag."
- You can change the icons used for required and changed notifications using skins. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

- You can make your input components accessible. For more information, see Chapter 33, "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. For more information, see Chapter 32, "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). For more information, see the "Creating a Basic Databound Page" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

11.2 Defining Forms

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.

Example 11–1 shows two input components and a Submit button that when clicked will submit both input values for processing.

Example 11–1 ADF Faces Form as a Container for Input Components

```
<af:form id="f1">
  <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: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 more information about subforms, see Section 5.5, "Using Subforms to Create Sections on a Page."

Example 11–2 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.

Example 11–2 ADF Faces Subform Within a Form

```
<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.

11.2.1 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. For more information, see Section 11.2, "Defining Forms."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 more information about uploading files, see Section 11.9, "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`.

11.2.2 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. For more information, see Section 11.2, "Defining Forms."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "Additional Functionality for Input Components and Forms."

You must add a form component to the page. For procedures, see Section 11.2.1, "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 more information about lifecycle phases, see Chapter 5, "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.

11.2.3 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. For more information, see Section 11.2, "Defining Forms."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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**.

11.3 Using the inputText Component

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 Section 11.8, "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

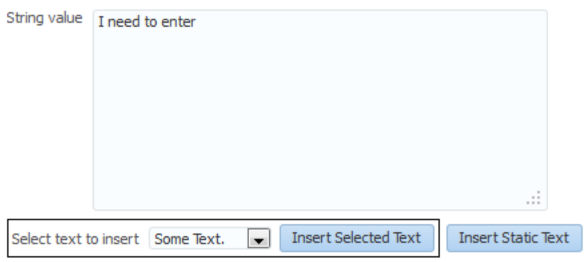
You can make the `inputText` component display more than one row of text using the `rows` attribute. If you set the `rows` attribute to be greater than one, and you set the `simple` attribute to true, then the `inputText` component can be configured to stretch to fit its container using the `dimensionsFrom` attribute. For more information about how components stretch, see Section 9.2.1, "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

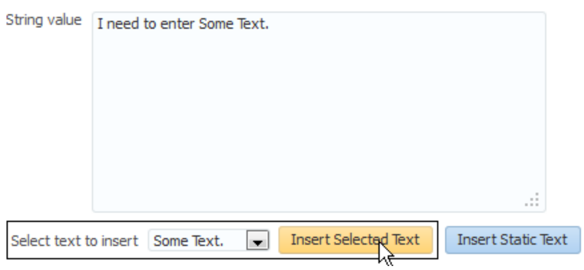
You can also configure an `insertTextBehavior` tag that works with command components to insert given text into an `inputText` component. The text to be entered can be a simple string, or it can be the value of another component, for example the selected list item in a `selectOneChoice` component. For example, Figure 11–10 shows an `inputText` component with some text already entered by a user.

Figure 11–10 inputText Component with Entered Text



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



11.3.1 How to Add an inputText Component

You can use an `inputText` component inside any of the layout components described in Chapter 9, "Organizing Content on Web Pages."

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 11.3, "Using the inputText Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 more information about using resource bundles, see Chapter 32, "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 more information about conversion, see Chapter 7.4, "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.

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 more information about using the `autoSubmit` attribute, see Section 5.3, "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. For more information, see Section 7.3, "Adding Conversion."
- **Validator**: Specify a method reference to a validator method using an EL expression. For more information, see Section 7.5, "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.

11.3.2 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. For more information, see Section 6.4, "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. For more information, see Section 11.3, "Using the inputText Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "Additional Functionality for Input Components and Forms."

Before you add an `insertTextBehavior` tag, you need to create an `inputText` component as described in Section 11.3.1, "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 Section 20.3.1, "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. Example 11–3 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.

Example 11–3 Using the insertTextBehavior Tag

```
<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.

11.4 Using the Input Number 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

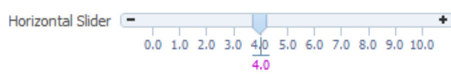
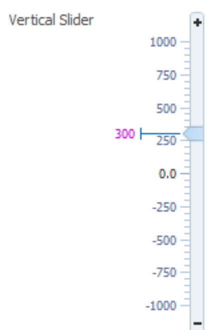
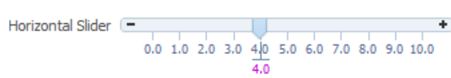


Figure 11–13 InputNumberSlider in Vertical Layout



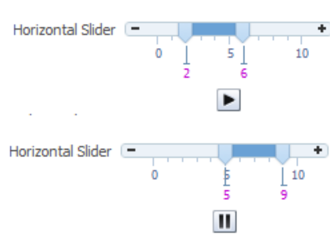
The `inputRangeSlider` component has two markers and allows the user to pick the end points of a range, as shown in Figure 11–14.

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

11.4.1 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. For more information, see Section 11.4, "Using the Input Number Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 Section 11.3.1, "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`.

11.4.2 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. For more information, see Section 11.4, "Using the Input Number Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "Additional Functionality for Input Components and Forms."

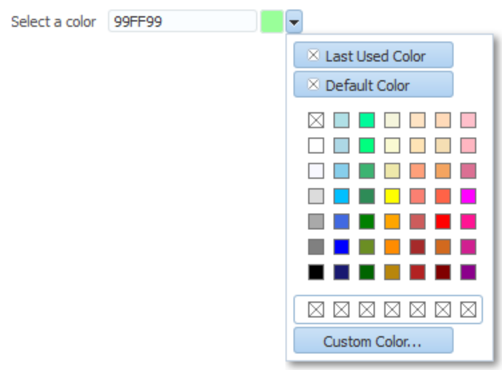
To add an inputNumberSpinbox component:

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 more information about setting these attributes, see Section 11.3.1, "How to Add an inputText Component."

11.5 Using Color and Date Choosers

The `inputColor` component allows users to pick a color from a 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



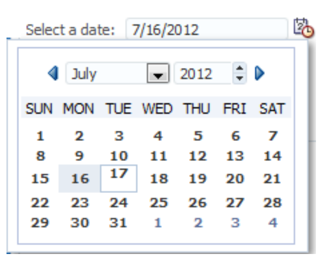
By default, the content delivery for the popup is lazy. When the user clicks the button, the `inputColor` component receives a PPR request, and rerenders, displaying a `chooseColor` component in a `popup` component.

> **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.

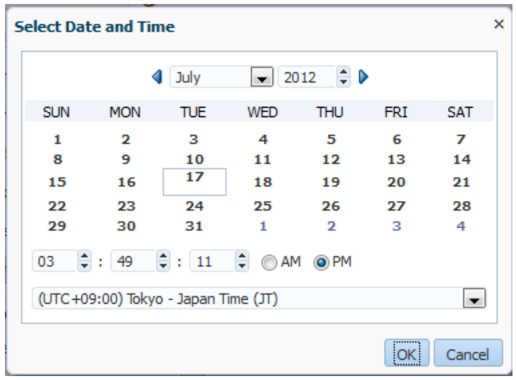
The `inputDate` component presents a text input field for entering dates and a button for picking dates from a popup calendar, as shown in Figure 11–18. The default date format is the short date format appropriate for the current locale. For example, the default format in American English (ENU) is `mm/dd/yy`. However, you can override the format using a date-time converter (for more information about using converters, see Section 7.3, "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



11.5.1 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:

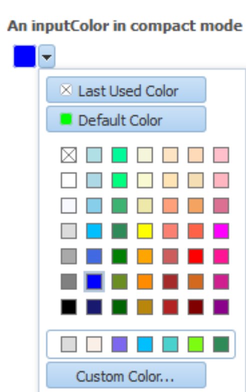
It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 11.5, "Using Color and Date Choosers."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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



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.
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:

- **ChooseId**: 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.

11.5.2 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. For more information, see Section 11.5, "Using Color and Date Choosers."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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. 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.
4. 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.

5. 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.
6. 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.
7. Optionally expand the Data section and set the **TimeZoneList** attribute to a **custom list of timezones**.

11.5.3 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 Example 11–4, 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.

Example 11–4 Using an inputDate and Time Zone Selection List Together

```
<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);
}
```

11.5.4 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.

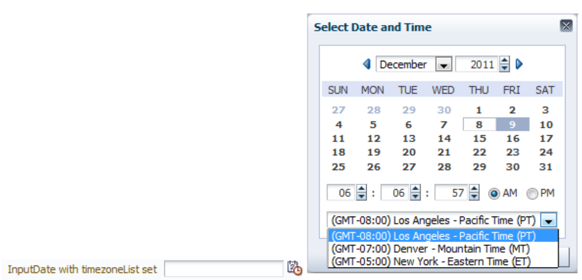
In Example 11–5, the `inputDate` component uses the custom time zone list defined by the `testInput` backing bean to show only three US time zones.

Example 11–5 Using Custom Time Zone List

```
<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 in Example 11–5.

Figure 11–21 Custom Time Zone List in inputDate Component



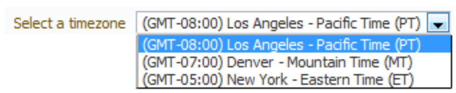
You can also use the `getCustomTimeZoneSelectItems` helper EL method to get custom time zones as a list of `SelectItems`, which can then be used with components like `selectOneChoice`, as shown in Example 11–6.

Example 11–6   Using getCustomTimeZoneSelectItems method To Access Custom Time Zone List

```
<af:selectOneChoice label="Select a timezone" id="soctzlel" 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 in Example 11–6.

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.

11.6 Using Selection Components

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

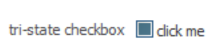
All the selection components except the `selectItem` component delivers the `ValueChangeEvent` and `AttributeChangeEvent` events. The `selectItem` component only delivers the `AttributeChangeEvent` event. You must create a `valueChangeListener` handler or an `attributeChangeListener` handler, or both 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

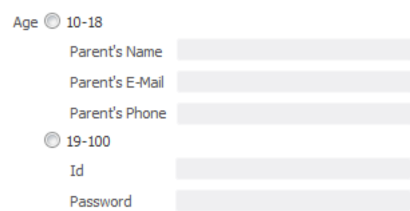


The `selectBooleanRadio` component displays a boolean choice, and must always be set to a boolean. Unlike the `selectBooleanCheckbox` component, the `selectBooleanRadio` component allows you to group `selectBooleanRadio` components together using the same `group` attribute.

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.

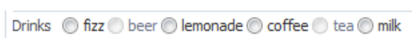
> **Tip:** Each `selectBooleanRadio` component must be bound to a unique boolean.

Figure 11–25 selectBooleanRadio Component



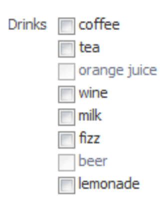
You use the `selectOneRadio` component to create a list of radio buttons from which the user can select a single value from a list, as shown in Figure 11–26.

Figure 11–26 selectOneRadio Component



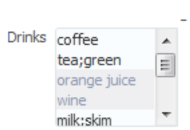
You use the `selectManyCheckbox` component to create a list of checkboxes from which the user can select one or more values, as shown in Figure 11–27.

Figure 11–27 selectManyCheckbox Component



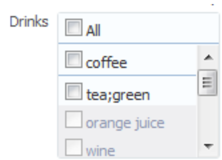
The `selectOneListbox` component creates a component which allows the user to select a single value from a list of items displayed in a shaded box, as shown in Figure 11–28.

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

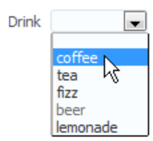


The `selectOneChoice` component creates a menu-style component, which allows the user to select a single value from a dropdown list of items. The `selectOneChoice` component is intended for a relatively small number of items in the dropdown list.

> **Best Practice:** If a large number of items is desired, use an `inputComboboxListOfValues` component instead. For more information, see Chapter 13, "Using List-of-Values Components."

The `selectOneChoice` component is shown in Figure 11–30.

Figure 11–30 selectOneChoice Component

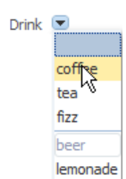


You can configure the `selectOneChoice` component to display in a compact mode, as shown in Figure 11–31. When in compact mode, the input field is replaced with a smaller icon.

Figure 11–31 selectOneChoice Component in Compact Mode

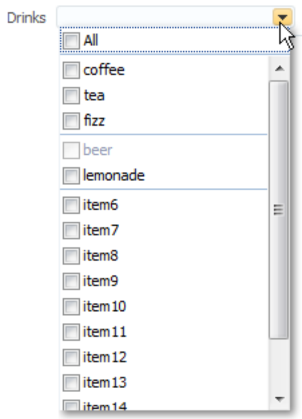
When the user clicks the icon, the dropdown list is displayed, as shown in Figure 11–32.

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 more information about content delivery for the `selectManyChoice` component and its limitations, see Section 11.6.2, "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 (for more information, see Section 4.5.1, "How to Set Property Values on the Client"):

- `selectOneChoice`
- `selectOneRadio`
- `selectOneListbox`
- `selectBooleanRadio`
- `selectBooleanCheckbox`
- `selectManyChoice`
- `selectManyCheckbox`
- `selectManyListbox`

11.6.1 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. For more information, see Section 11.6, "Using Selection Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 more information about messages, see Section 19.3, "Displaying Hints and Error Messages for Validation and Conversion."
   - **Validator**: Enter an EL expression that resolves to a validation method on a managed bean (for more information, see Chapter 7, "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. For more information, see Section 11.3.1, "How to Add an inputText Component."

Table 11–1 Appearance Attributes for Selection Components

Components	Attribute
`selectOneRadio`, `selectManyCheckbox`	**Layout**: Set to `vertical` to have the buttons or checkboxes displayed vertically. Set to `horizontal` to have them displayed in a single horizontal line.
`selectManyListbox`	**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 `size` attribute value, a scrollbar will be displayed.
`selectManyListbox`, `selectManyChoice`	**SelectAllVisible**: Set to `true` to display an **All** selection that allows the user to select all items in the list.
`selectOneChoice`	**Mode**: Set to `compact` to display the component only when the user clicks the dropdown icon.
`selectOneRadio`, `selectOneListbox`, `selectOneChoice`	**UnselectedLabel**: Enter text for the option that represents a value of `null`, meaning nothing is selected. If `unselectedLabel` 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 `required` attribute to `true` when defining an `unselectedLabel` value. If you do not, two blank options will appear in the list. Once an option has been successfully selected, and if `unselectedLabel` is not set, then the empty option will not be rendered.

4. 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. For more information, see Section 11.3.1, "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 `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`. 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 `selectBooleanRadio` components with the same `group` value.

5. 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 more information about content delivery for the `selectManyChoice` component, see Section 11.6.2, "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.

11.6.2 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:

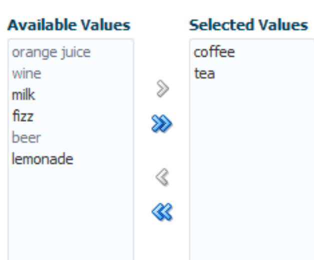
- You cannot store the value of the selectManyChoice in Request scope. On postback, the value attribute is accessed from the model, rather than decoding what was returned from the client. If the value is stored in Request scope, that value will be empty. Do not store the value in Request scope.
- On postbacks, converters are not called. If you are relying on converters for postbacks, then you should not use lazy content delivery.
- The contentDelivery attribute is ignored when in screen reader mode. The selectItem components will always be built when the page is rendered.

11.7 Using Shuttle Components

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.

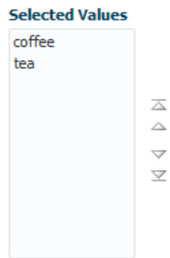
Similar to other select components, the List or Array of items are composed of `selectItem` components nested within the `selectManyShuttle` or `selectOrderShuttle` component. Example 11–7 shows a sample `selectOrderShuttle` component that allows the user to select the top five file types from a list of file types.

Example 11–7 selectOrderShuttle JSF Page Code

```
<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



11.7.1 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. For more information, see Section 11.7, "Using Shuttle Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 more information about messages, see Section 19.3, "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.

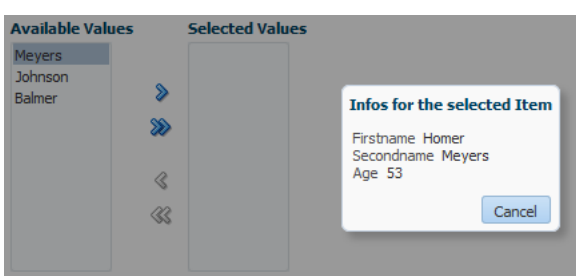
4. Expand the Behavior section and optionally set the following attributes:
   - **ValuePassThru**: Specify whether or not the values are passed through to the client. When `valuePassThru` is `false`, the value and the options' values are converted to indexes before being sent to the client. Therefore, 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.

11.7.2 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 more information about using client listeners for events, see Section 4.4, "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 Example 11–8.

Example 11–8 Using a clientListener to Register a Selection

```
<af:selectManyShuttle value="#{demoInput.manyListValue1}" id="sms1"
     valuePassThru="true" ...>
     <af:clientListener type="propertyChange" method="showDetails"/>
     <af:selectItem label="coffee" value="bean" id="si1" />
      ...
</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 Example 11–9, `AdfShuttleUtils.getLastSelectionChange` is called to get the value of the last selected item

Example 11–9 Sample JavaScript methods showDetails used to process a selection

```
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
```

```
        }
    }
}
```

11.8 Using the richTextEditor Component

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

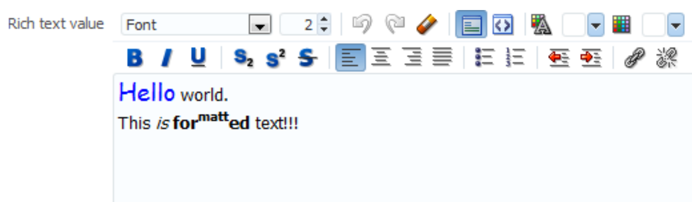
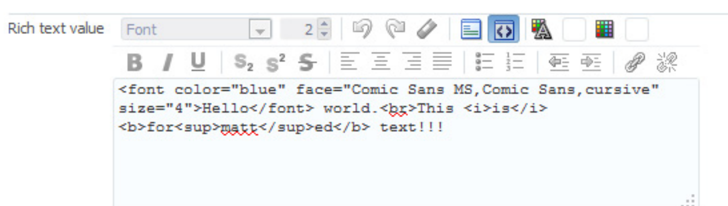


Figure 11–39 shows the editor in source code editing mode.

Figure 11–39 The 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

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 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 11–40.

Figure 11–40 Toolbar in richTextEditor Component

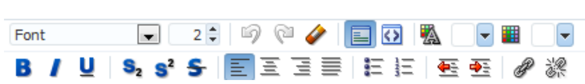
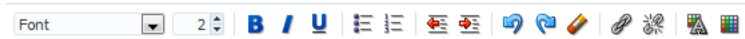


Figure 11–41 shows a toolbar that has been customized. Many of the toolbar buttons have been removed and a toolbar with a custom toolbar button and a menu have been added.

Figure 11–41 Customized Toolbar for richTextEditor



11.8.1 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. For more information, see Section 11.8.2, "How to Add the Ability to Insert Text into a richTextEditor Component," and Section 11.8.3, "How to Customize the Toolbar."

Before you begin:

It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 11.8, "Using the richTextEditor Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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. For more information, see Section 12.2.2, "Content Delivery."

     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`. For more information, see Section 9.2.1, "Geometry Management and Component Stretching."

11.8.2 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. For more information, see Section 11.8, "Using the richTextEditor Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "Additional Functionality for Input Components and Forms."

You need to create a `richTextEditor` component as described in Section 11.8.1, "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 Section 20.3.1, "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 Example 11–10 shows preformatted text as the value for an attribute in the `demoInput` managed bean.

Example 11–10 Preformatted Text in a 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 &amp; Sunday 'til 5:00 pm</font>" +
  "</p>";
```

Example 11–11 shows how the text is referenced from the `richTextEditorInsertBehavior` tag.

Example 11–11 Using 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.

11.8.3 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. For more information, see Section 11.8, "Using the richTextEditor Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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 11–41, 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 more information about toolbars and toolbar buttons, see Section 16.3, "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 `toolboxLayout` 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.

11.9 Using File Upload

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 Section 20.5.1, "How to Use an Action Component to Download Files").

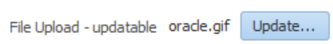
The `inputFile` component delivers the standard `ValueChangeEvent` event as files are being uploaded, and it manages the loading process transparently. The `value` property of an `inputFile` component is set to an instance of the `org.apache.myfaces.trinidad.model.UploadedFile` class when the file is 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 11–42, that can be used to upload a file.

Figure 11–42 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 11–43. This will allow the user to modify the value of the `inputFile` component.

Figure 11–43 inputFile Component in Update Mode



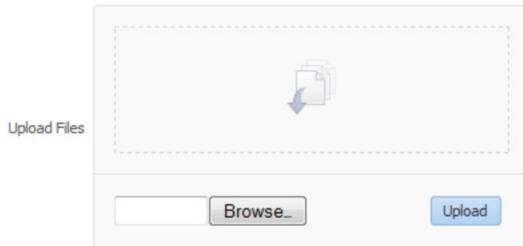
Note: When the file is uploaded, the value of the `inputFile` component becomes the instance of the `org.apache.myfaces.trinidad.model.UploadedFile` class. Therefore, if you need to access the value (that is, the file itself), you need to access this class. Accessing the component itself through the `binding` attribute only accesses the component and not the uploaded file.

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 11–44.

Figure 11–44 inputFile Component in Read-Only Mode

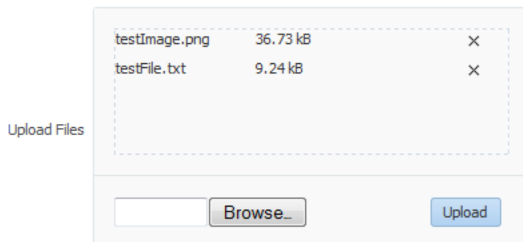
By default, the `inputFile` component allows upload of one file only, but it can be configured to upload multiple files. Figure 11–45 shows the `inputFile` component configured to upload multiple files.

Figure 11–45 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 11–46.

Figure 11–46 inputFile Component Showing Files Ready to Upload



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. The API lets you get at the actual byte stream of the file, as well as the file's name, its MIME type, and its size.

Note: 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 11–12.

Example 11–12 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>

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()
    }
  }
}
```

You can also handle the upload by binding the value directly to a managed bean, as shown in Example 11–13.

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;

}
```

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`.

11.9.1 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. For more information, see Section 11.9, "Using File Upload."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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.

11.9.2 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. For more information, see Section 11.9, "Using File Upload."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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
`submit`	Upload one file only. The drop section, where the user can drag-and-drop files, is not displayed.
`auto`	Show the drop section and enable upload of multiple files. The upload starts immediately when the files appear in drop section. If `maximumFiles` is set to 1, the user can upload multiple files by selecting one file at a time, instead of selecting multiple files together.
`manual`	Show the drop section and enable upload of multiple files. The upload starts when the **Upload** button is clicked. If `maximumFiles` is set to 1, the user can upload multiple files by selecting one file at a time, instead of selecting multiple files together.
`autoIfMultiple`	Upload multiple files. The upload starts immediately when the files appear in the drop section. By default, `uploadType` is set to `autoIfMultiple`. If `maximumFiles` is set to 1, the user can select and upload one file only. The drop section is also not displayed.
`manualIfMultiple`	Upload multiple files. The upload starts when the **Upload** button is clicked. If `maximumFiles` 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

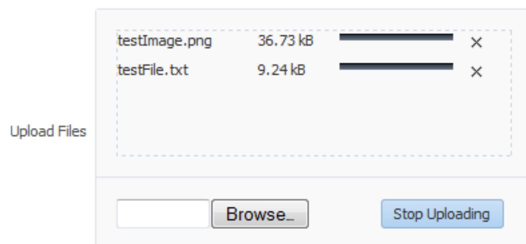
enables the display of multiple file upload user interface. The valid values are `default`, `dropArea`, and `none`.

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.

For more information, see Section 11.9.5, "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 11–47.

Figure 11–47 File Being Uploaded Using inputFile Component



11.9.3 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 Example 11–14.

Example 11–14 Parameters That Define File Upload Size and Directory

```
<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>
```

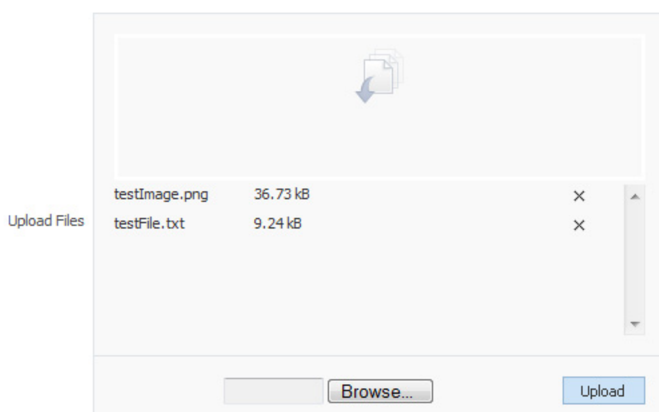
You can customize the file upload process by replacing the entire `org.apache.myfaces.trinidad.webapp.UploadedFileProcessor` class with the `<uploaded-file-processor>` element in the `trinidad-config.xml` configuration file. Replacing the `UploadedFileProcessor` class makes the parameters listed in Example 11–14 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.

11.9.4 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 11–48.

Figure 11–48 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.

Note: If you are using OracleAS Single Sign-On (SSO), you might need to configure the `mod_osso.conf` file that enables single sign-on for Oracle HTTP Server. The file is located at `ORACLEOHS_HOME/ohs/conf/`, where `ORACLEOHS_HOME` refers to the home directory of the Oracle HTTP server installation. The configuration is required for the upload applet to function properly in non-HTML5 browsers.

Update the `mod_osso.conf` file with the following parameters:

```
OssoSecureCookies off
OssoHTTPOnly Off

Header unset Pragma
OssoSendCacheHeaders off
```

For more information about OracleAS Single Sign-On, see *Securing Applications with Oracle Platform Security Services*.

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 more information, see *Oracle JDeveloper Release Notes* on Oracle Technology Network.

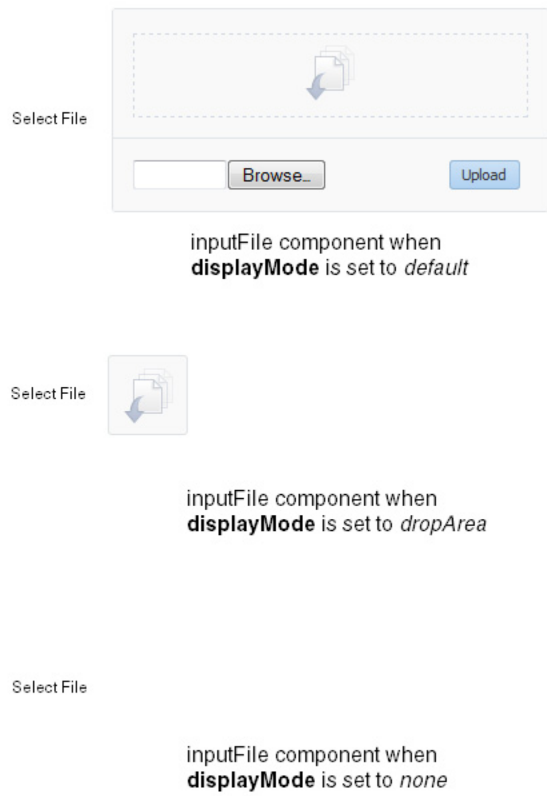
After uploading all files, you must ensure that the form is submitted, else the `inputFile` component data will not be uploaded to the server. If `autoSubmit` is set to `true` on the `inputFile` component, then the form is submitted automatically after all the files have finished uploading. After the 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 11–12.

11.9.5 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 11–49 shows the `inputFile` component with different values for `displayMode`.

Figure 11–49 displayMode Attribute Values for inputFile Component



Example 11–15 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.

Example 11–15 Customized inputFile Component With Input Fields and Buttons

```
<af:inputFile binding="#{editor.component}" id="testid" immediate="true"
              maximumFiles="-1"valueChangeListener="#{demoFile.fileUpdate}"
              displayMode="dropArea" uploadType="manual"contentStyle="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: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 11–50 shows the `inputFile` component of Example 11–15 at runtime.

Figure 11–50 Customized inputFile Component

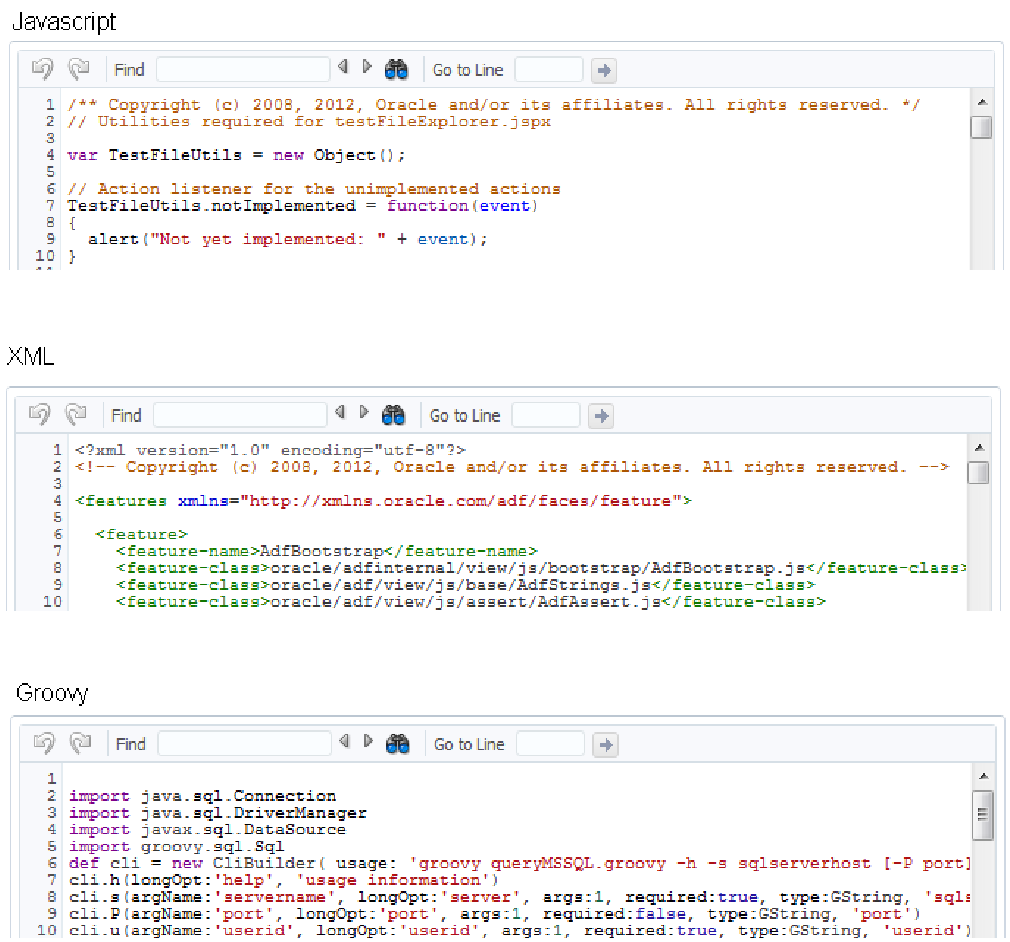
You can use the JavaScript API methods to customize the behavior of `inputFile` component. For more information, see the *JavaScript API Reference for Oracle ADF Faces*.

11.10 Using Code Editor

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, basic validation, highlighting errors, and a message pane for logs. Using the code editor, the user won't need to run an IDE software to test a program code for errors or warnings.

The code editor component supports Javascript, XML, and Groovy languages, as shown in Figure 11–51.

Figure 11–51 Code Editor Component using Javascript, XML, and Groovy



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
- Auto-indent
- Auto-format
- Message pane for error messages
- Support for large files with more than thousand lines of code

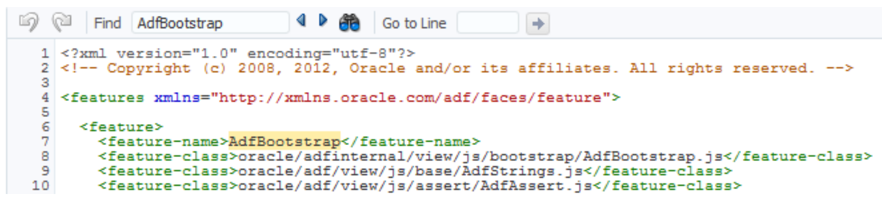
To add or edit code in code editor, click the editor area, and add your code. To forward-indent a line of code, press F2 to enable the edit mode of the editor and press Tab key. To backward-indent a line of code, press Shift+Tab keys. You must enable the edit mode to indent code, else pressing Tab or Shift+Tab would navigate to the next or the previous component.

The user can use the toolbar (shown in Figure 11–52) to undo and redo the changes, search and replace text, and jump to a specific line number.

Figure 11–52 Code Editor Toolbar

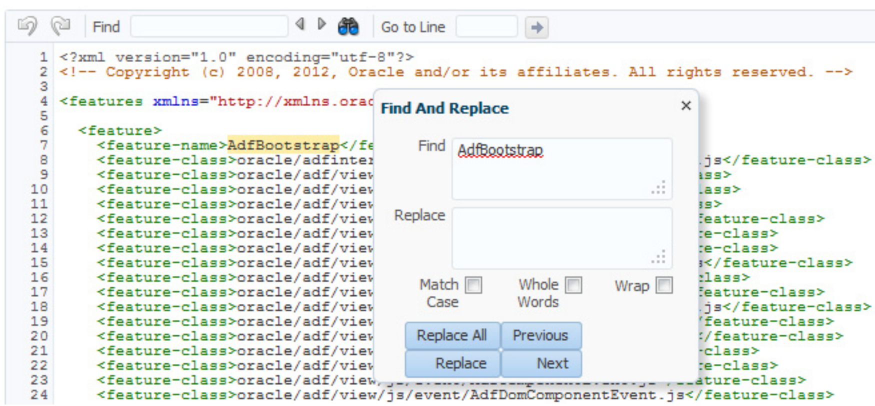
To search for a string, enter the search term in the **Find** field, and click **Find Next** or **Find Previous** icons to locate the search string in the code editor. Figure 11–53 shows the Find field of the toolbar used to search a string in the code editor.

Figure 11–53 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 11–54.

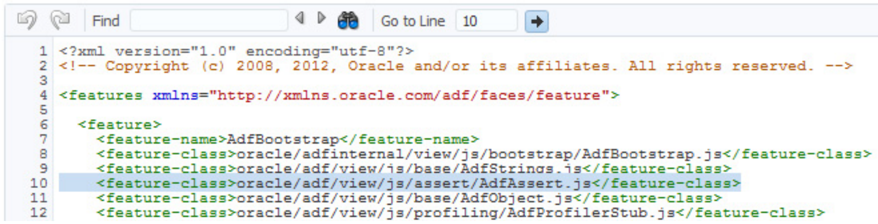
Figure 11–54 Using Find and Replace Dialog of Code Editor



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.

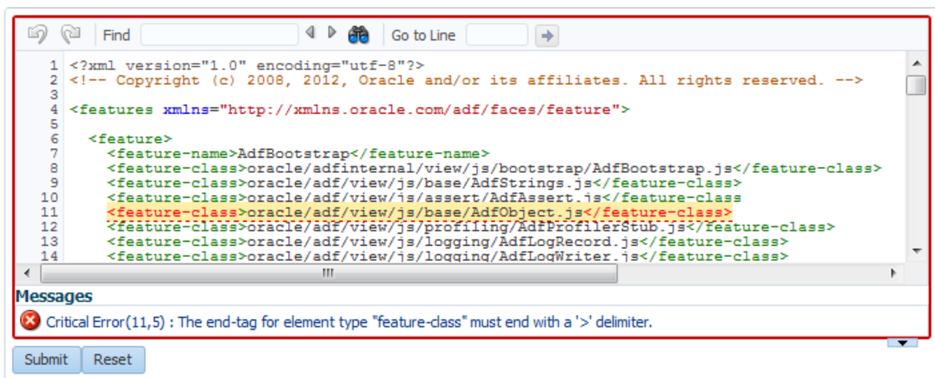
To jump to a specific line number, enter the number in the **Go to Line** field and click **Jump to line**, as shown in Figure 11–55.

Figure 11–55 Using Go To Line Feature of Code Editor



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 11–56 shows the message pane listing all XML errors noticed by the XML parser running on the server.

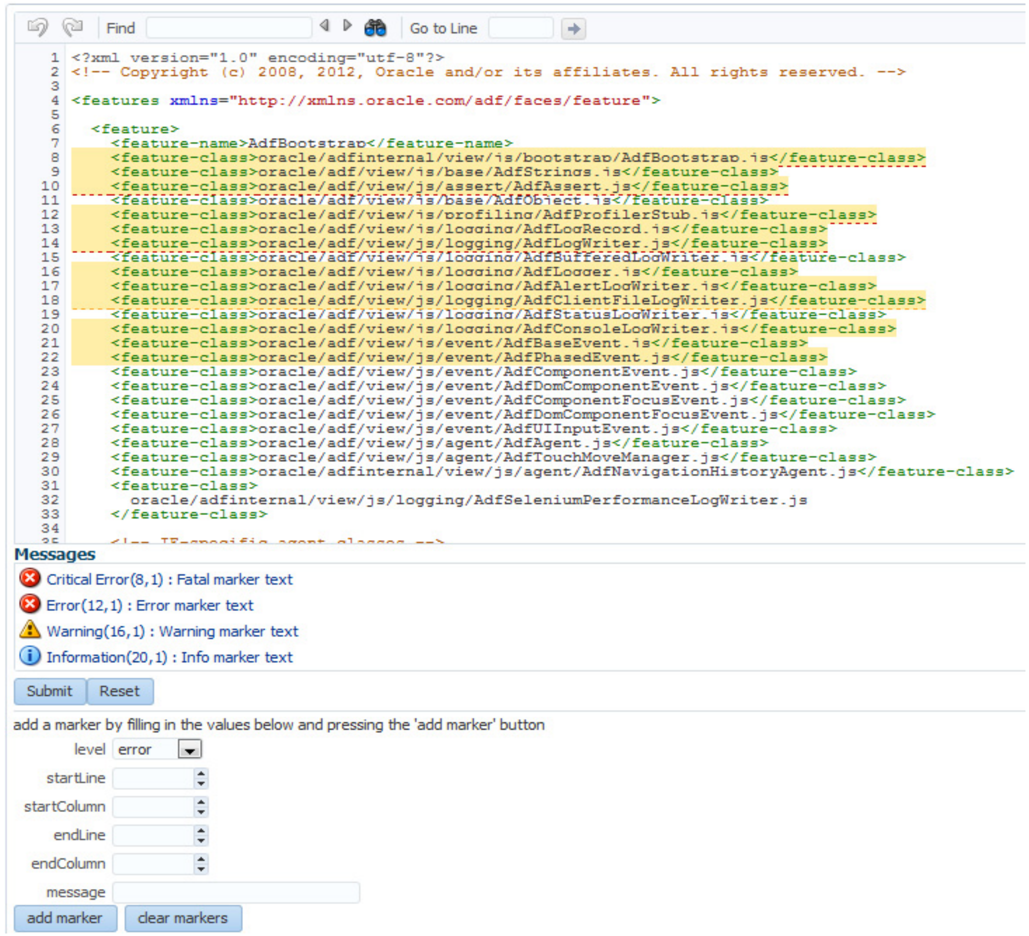
Figure 11–56 Message Pane of Code Editor



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 11–57 shows the code editor with error, critical, warning, and information markers.

Figure 11–57 Using Markers in Code Editor



11.10.1 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. For more information, see Section 11.10, "Using Code Editor."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 11.1.2, "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. For more information see the "Creating ADF Databound Tables," "Displaying Master-Detail Data" and "Using More Complex Databound ADF Faces Components" chapters of *Developing Fusion Web Applications with Oracle Application Development Framework*

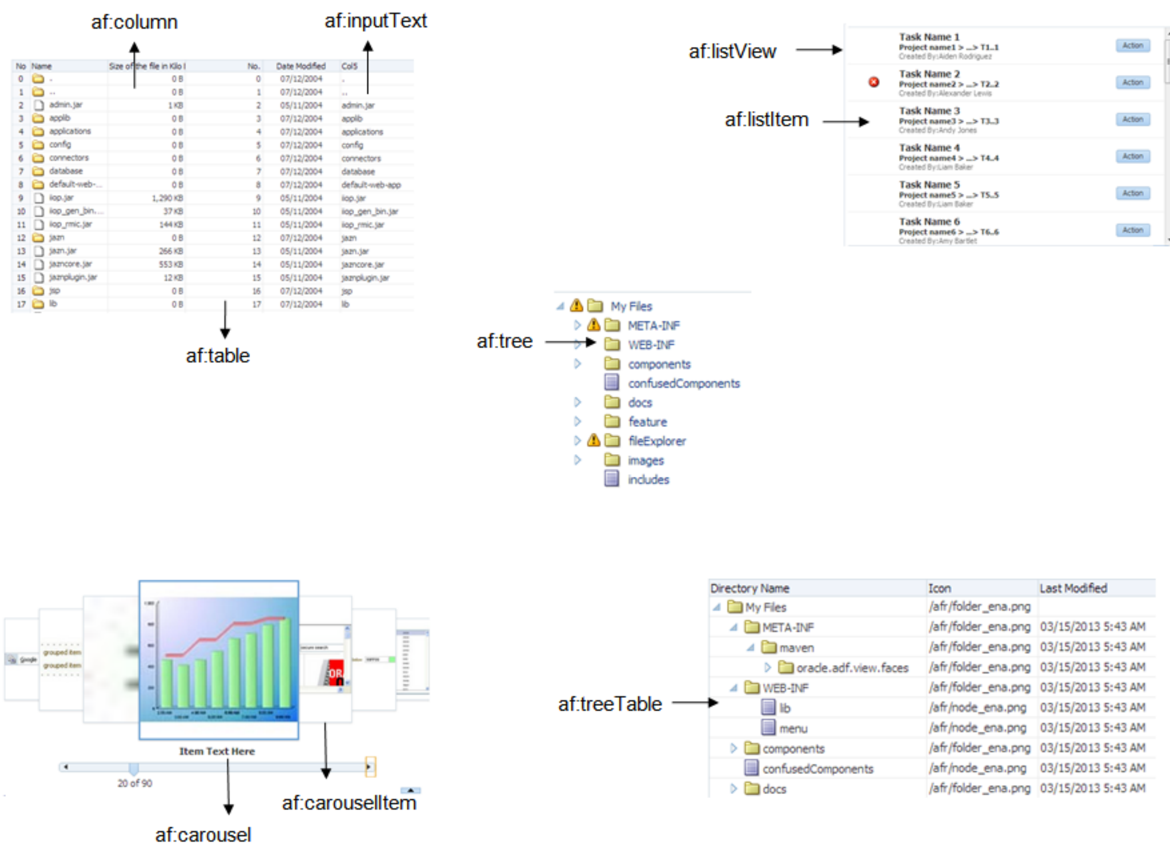
This chapter includes the following sections:

12.1 About Collection-Based 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

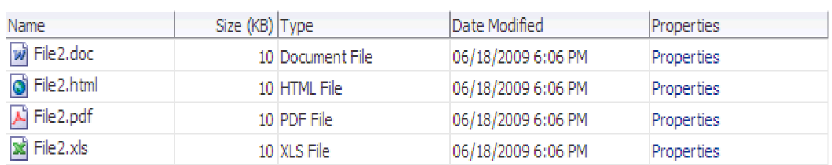


> **Tip:** When you do not want to use a table, but still need the same stamping capabilities, you can use the iterator tag. For example, say you want to display a list of periodic table elements, and for each element, you want to display the name, atomic number, symbol, and group. You can use the iterator 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.

12.1.1 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.

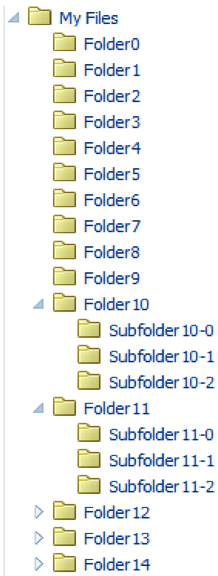
Figure 12–2 Table Component in the File Explorer Application



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

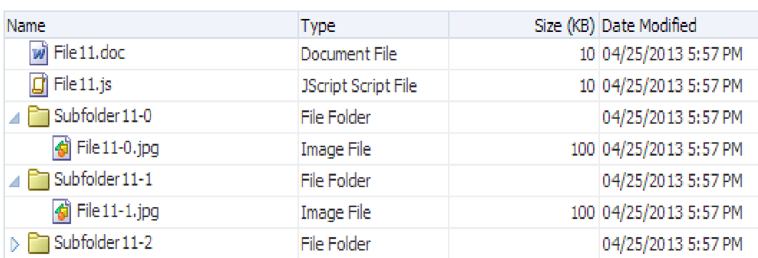
Hierarchical data (that is data that has parent/child relationships), such as the directory in the File Explorer application, can be displayed as expandable trees using the tree component. Items are displayed as nodes that mirror the parent/child structure of the data. Each top-level node can be expanded 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



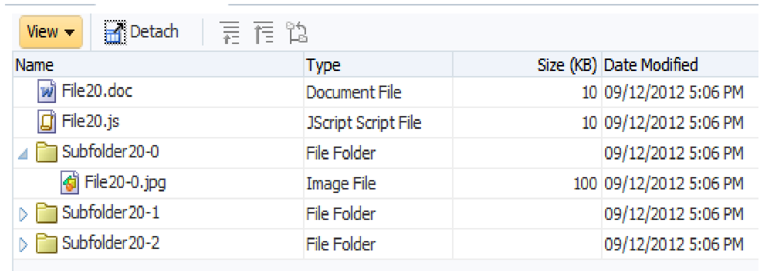
Name	Type	Size (KB)	Date Modified
File11.doc	Document File	10	04/25/2013 5:57 PM
File11.js	JScript Script File	10	04/25/2013 5:57 PM
Subfolder11-0	File Folder		04/25/2013 5:57 PM
File11-0.jpg	Image File	100	04/25/2013 5:57 PM
Subfolder11-1	File Folder		04/25/2013 5:57 PM
File11-1.jpg	Image File	100	04/25/2013 5:57 PM
Subfolder11-2	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



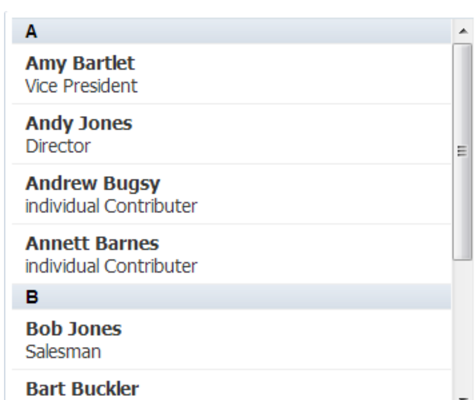
The `listView` component is a light-weight table that allows you to display structured data in a list format. Unlike a table, it does not have columns, which allows you to easily present data in a variety of patterns, beyond a simple tabular layout.

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

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

12.1.2 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 more information about menus, toolbars, and toolbar buttons, see Chapter 16, "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 more information about how components stretch, see Section 9.2.1, "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. For more information, see the "Using the Active Data Service" chapter of *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. For more information, see Chapter 6, "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. For more information, see Section 5.3, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- **Accessibility**: You can make your components accessible. For more information, see Chapter 33, "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. For more information, see the "Creating ADF Databound Tables" and "Displaying Master-Detail Data" chapters of *Developing Web User Interfaces with Oracle ADF Faces*.

12.2 Common Functionality in Collection-Based Components

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.

12.2.1 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 more information about how stamping works, especially with client components, see Section 12.2.6, "Accessing Client Collection Components."

Example 12–1 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.

Example 12–1 JSF Code for a Table Uses the var Attribute to Access Values

```
<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 more 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. For more information see the "Creating ADF Databound Tables" chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

12.2.2 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).

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. If it is not available, the data is loaded during the separate PPR request, as it is with lazy delivery.

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.

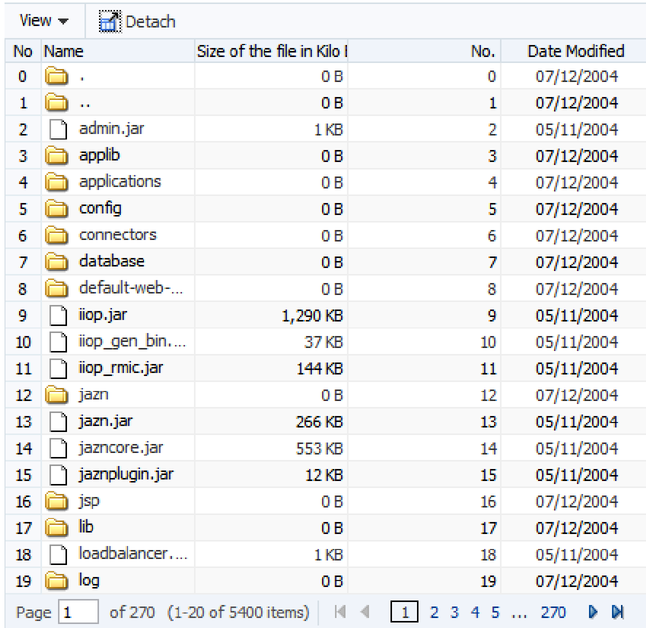
When available delivery provides the additional flexibility of using immediate when data is available during initial rendering or falling back on lazy when data is not initially available.

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.

By default, on a desktop device, tables render a scroll bar that allows the users to scroll through the rows of data. Instead, you can configure the table to be paginated using the `scrollPolicy` attribute, so that it 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



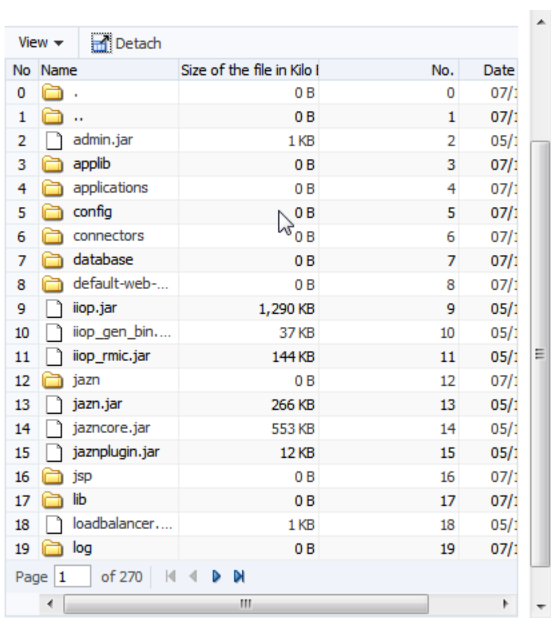
View ▾ | Detach

No	Name	Size of the file in Kilo I	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 Paginated Table in Compact Mode



View ▾ | Detach

No	Name	Size of the file in Kilo I	No.	Date
0	.	0 B	0	07/
1	..	0 B	1	07/
2	admin.jar	1 KB	2	05/
3	applib	0 B	3	07/
4	applications	0 B	4	07/
5	config	0 B	5	07/
6	connectors	0 B	6	07/
7	database	0 B	7	07/
8	default-web-...	0 B	8	07/
9	iiop.jar	1,290 KB	9	05/
10	iiop_gen_bin....	37 KB	10	05/
11	iiop_rmic.jar	144 KB	11	05/
12	jazn	0 B	12	07/
13	jazn.jar	266 KB	13	05/
14	jazncore.jar	553 KB	14	05/
15	jaznplugin.jar	12 KB	15	05/
16	jsp	0 B	16	07/
17	lib	0 B	17	07/
18	loadbalancer....	1 KB	18	05/
19	log	0 B	19	07/

Page 1 of 270

Note: By default, tables are rendered to display as paginated on tablet devices.

In order for a table to display as paginated, you must set the `scrollPolicy` attribute to `page`, the `autoHeight`Rows attribute to `0`, and the table must be placed in a flowing container (that is, a component that does not stretch its children). For more information about flowing container components, see Section 9.2.1, "Geometry Management and Component Stretching."

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 scrollbars, on iOS operating systems, the scrollbars 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. For more information, see Section A.2.3.30, "Scrollbar Behavior in Tables."

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.

12.2.3 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. For more information, see Section 12.3.8, "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.

12.2.4 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).

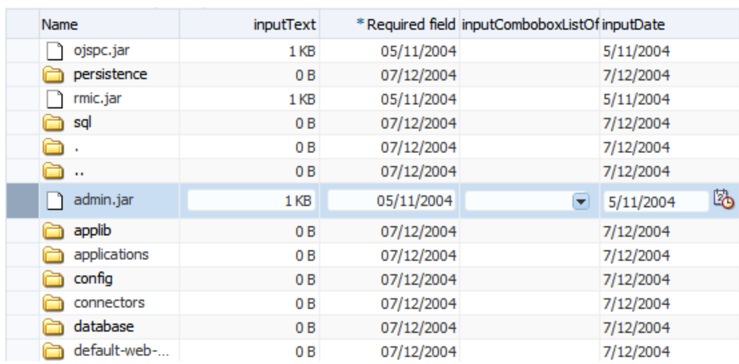
When you decide to use components whose value can be edited to display your data, you have the option of having the table, tree, or tree table either display all rows as available for editing at once, or display all but the currently active row as read-only using the `editingMode` attribute. For example, 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	select all	inputColor
0	.	0 B	7/12/2004	1979	Cool	99FF99
1	..	0 B	7/12/2004	1979	Cool	99FF99
2	admin.jar	1 KB	5/11/2004	1979	Cool	99FF99
3	applib	0 B	7/12/2004	1979	Cool	99FF99
4	applications	0 B	7/12/2004	1979	Cool	99FF99
5	config	0 B	7/12/2004	1979	Cool	99FF99
6	connectors	0 B	7/12/2004	1979	Cool	99FF99
7	database	0 B	7/12/2004	1979	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. 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.

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		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 singe 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.

12.2.5 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 more information about creating context menus, see Section 15.2, "Declaratively Creating Popups."

12.2.6 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 more information about peers, see Section 4.1, "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`.

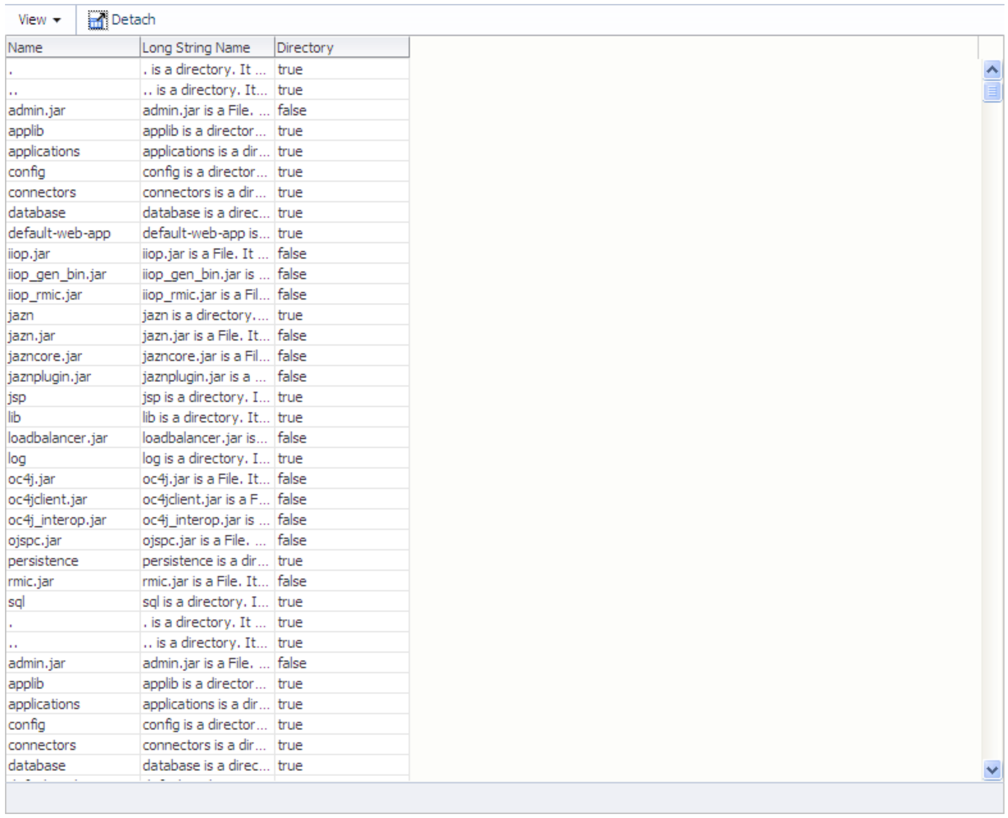
Because there is no client-side support for EL in the ADF Faces framework, nor is there support for sending entire table models to the client, the client-side code cannot rely on component stamping to access the value. Instead of reusing the same component instance on each row, a new JavaScript client 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 Section 12.13, "Accessing Selected Values on the Client from Collection-Based Components."

12.2.7 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.

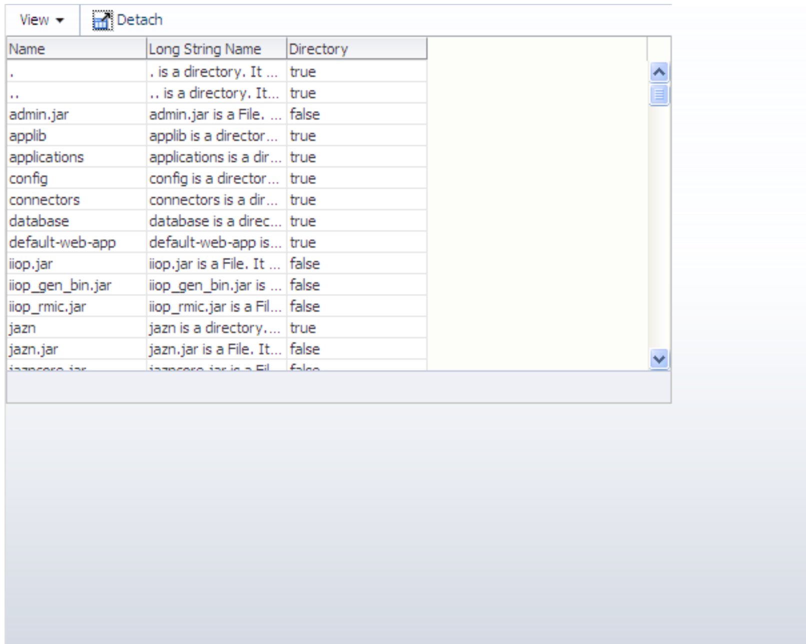
Figure 12–13 Table Stretches But Columns Do Not



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
jazn	jazn is a directory....	true
jazn.jar	jazn.jar is a File. It...	false
jazncore.jar	jazncore.jar is a Fil...	false
jaznplugin.jar	jaznplugin.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



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). For more information, see Section A.2.3.29, "Geometry Management for Layout and Table Components."

12.3 Displaying Data in Tables

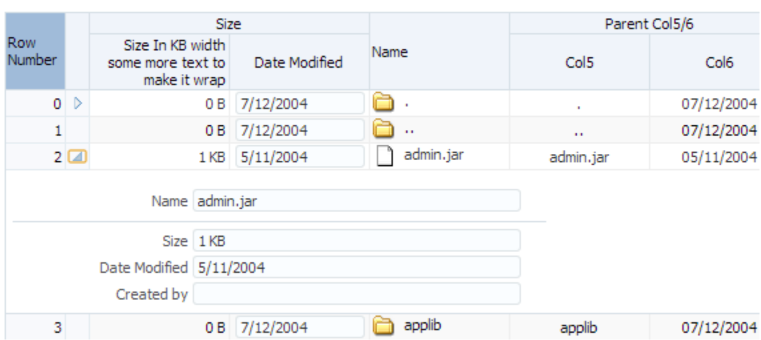
The immediate children of a table component must be `column` components. Each visible column component is displayed as a separate column in the table. Column components contain components used to display content, images, or provide further functionality. For more information about the features available with the column component, see Section 12.3.1, "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 Section 12.3.9, "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.

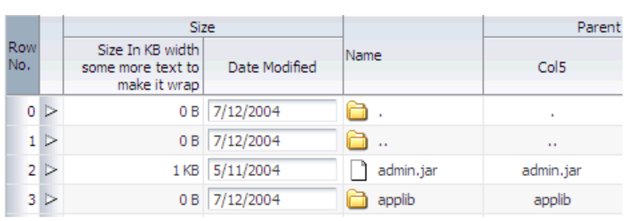
Figure 12–15 Extra Data Can Be Optionally Displayed



Row Number		Size: Size In KB width some more text to make it wrap	Size: Date Modified	Name	Parent Col5/6: Col5	Parent Col5/6: 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
		Name	admin.jar			
		Size	1 KB			
		Date Modified	5/11/2004			
		Created by				
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: Size In KB width some more text to make it wrap	Size: Date Modified	Name	Parent: 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 more information about using the `detailStamp` facet, see Section 12.4, "Adding Hidden Capabilities to a Table."

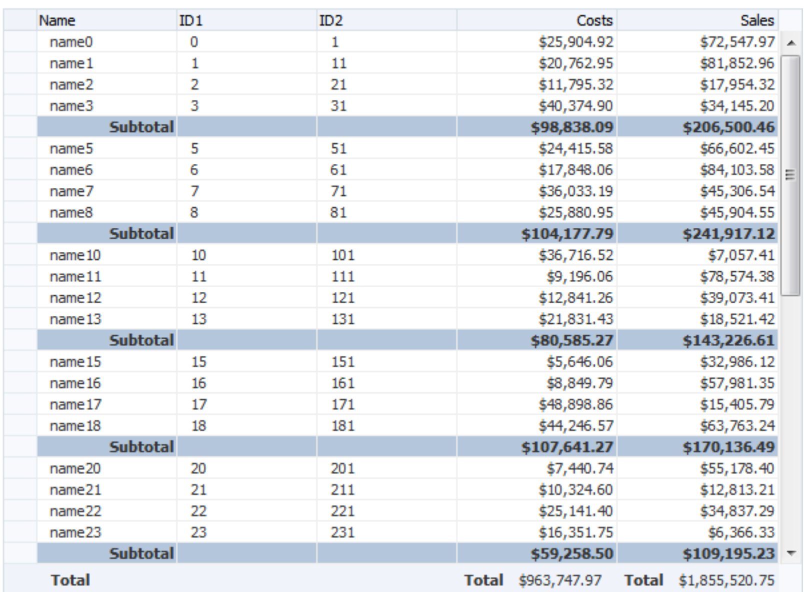
12.3.1 Columns and Column Data

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

12.3.2 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 programatically perform some action on the selected rows, see Section 12.3.8, "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. For more information, see Section 12.2.7, "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.
- 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. For more information, see Section 12.2.4, "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. For more information, see Section 12.9, "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.

12.3.3 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 special indicator on a column header lets the user know that the column can be sorted. 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.

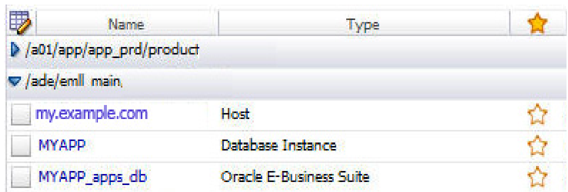
  By default, sorting is case-sensitive. That is, `abc` would be sorted before `ABC`. You can configure the column so that instead, `abc` would be sorted the same as `ABC`, using the `sortStrength` attribute.

  In order for the table to be able to sort, the underlying data model must also support sorting. For more information, see Section 12.3.7, "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.

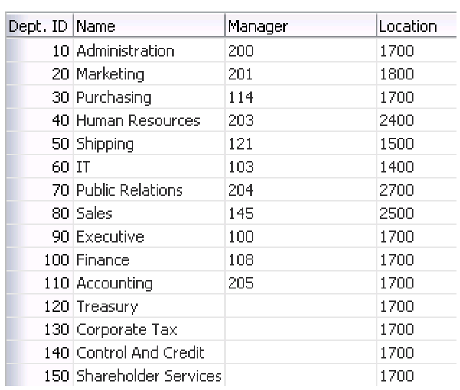
- Column width: The width of a column can be specified as an absolute value in pixels using the `width` attribute. If you configure a column to allow stretching, then you can also set the width as a percentage.
- Column spanning: You can configure a column to span across other columns using the `colSpan` 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



- 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.

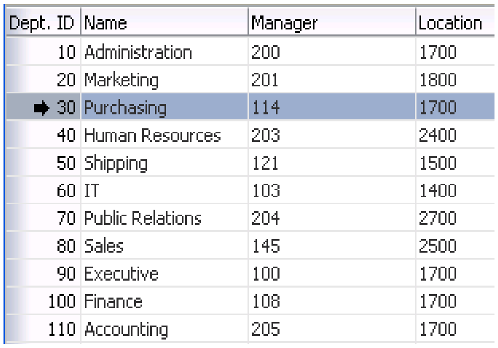
Figure 12–19 Row Header in a Table



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

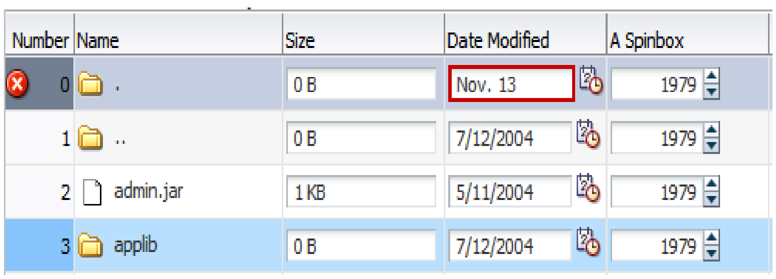


Dept. ID	Name	Manager	Location
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700

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 Chapter 35, "Allowing User Customization on JSF Pages."

12.3.4 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. For more information see the "Creating ADF Databound Tables" chapter of *Developing Web User Interfaces with Oracle ADF Faces*.

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. For more information, see Section 12.3, "Displaying Data in Tables."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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 more 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 more information about conversion, see Chapter 7.4, "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 programatically perform some action on the selected rows, see Section 12.3.8, "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:
   >
   > 1. In the Structure window, right-click the component created by the dialog and choose **Convert**.
   > 2. 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:
   >
   > 1. In the Structure window, right-click the first component and choose **Insert before** or **Insert after**. Select the component to insert.
   > 2. 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**.
   > 3. 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 more information about how components stretch, see Section 12.2.7, "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 Chapter 35, "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 more information, see Section 12.5, "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, the table is paginated, and displays a footer that allows the user to jump to specific pages of rows. Set the value to `auto` to keep this default behavior. Set the value to `scroll` to have the table always render a scroll bar. 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.

  Note: In order for a table to display as paginated, you must set the `scrollPolicy` attribute to `page`, the `autoHeightRows` attribute to `0`, and the table must be placed in a flowing container (that is, a component that does not stretch its children). If these conditions are not met, the table will display a scroll bar. For more information about flowing container components, see Section 9.2.1, "Geometry Management and Component Stretching."

- 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 Chapter 35, "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. For more information, see Section 12.9, "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. For information, see Chapter 35, "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. For more information, see Section 12.2.2, "Content Delivery."

- **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. For more information, see Section 12.2.2, "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 more information about stretching the table, see Section 12.2.7, "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 it's 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). For more information, see Section A.2.3.29, "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), or you want the user to click a row to make it editable (clickToEdit). For more information, see Section 12.2.4, "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 more information about context menus, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."
- **FilterModel**: Use in conjunction with filterVisible. For more information, see Section 12.5, "Enabling Filtering in Tables."
- Various listeners: Bind listeners to methods that will execute when the table invokes the corresponding event. For more information, see Chapter 6, "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 (for more information, 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 more information about skins, see Chapter 31, "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. For more information, see Section 12.3.7, "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. For more information, see Section 12.5, "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.

    Example 12–2 shows how you might set **colSpan** in a tree table component where you want only the parent node to span across all columns.

Example 12–2 Set colSpan to Span Parent Node to End of Table

```
<af:column id="c1" sortable="true" sortProperty="Dname"
           colSpan="#{testBean.container ? 'ALL' : '1'}"
           headerText="DepartmentName">
  <af:outputText value="#{node.Dname}" id="ot2"/>
</af:column>
```

Example 12–3 shows the corresponding managed bean code.

Example 12–3 Managed Bean Code to Span Columns

```
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**.

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 it neighbor columns.

    To allow the input component to be automatically sized when it is not the direct child of a column, set `contentStyle="width:auto"`.

12.3.5 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.

Example 12–4 shows abbreviated page code for the table in the File Explorer application.

Example 12–4 ADF Faces Table in the File Explorer Application

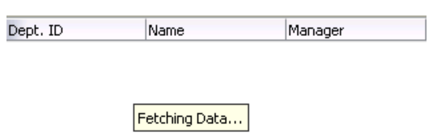
```
<af:table id="folderTable" var="file"
          value="#{explorer.contentViewManager.
                                  tableContentView.contentModel}"
           binding="#{explorer.contentViewManager.
                                  tableContentView.contentTable}"
           emptyText="#{explorerBundle['global.no_row']}"
           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:commandLink text="#{explorerBundle['global.properties']}"
                    partialSubmit="true"
                    action="#{explorer.launchProperties}"
                    returnListener="#{explorer.returnFromProperties}"
                    windowWidth="300" windowHeight="300"
                    useWindow="true">
    </af:commandLink>
  </af:column>
</af:table>
```

12.3.6 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.

Figure 12–22 Table Fetches Data in a Second PPR Request



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.

12.3.7 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.

For more information, 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. SortableModel is a concrete class that extends `CollectionModel` and implements all the sorting functionalities.

Note: Automatic support provides sorting for only one column. Multi-column sorting is not supported.

12.3.8 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.

Example 12–5 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.

Example 12–5 Selecting Rows

```
<af:table binding="#{mybean.table}" rowselection="multiple" ...>
  ...
</af:table>
<af:button text="Delete" actionListener="#{mybean.performDelete}"/>
```

Example 12–6 shows an actions method, `performDelete`, which iterates through all the selected rows and calls the `markForDeletion` method on each one.

Example 12–6 Using the rowKey Object

```
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);
  }
}
```

12.3.9 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 Example 12–7.

Example 12–7 Managed Bean Returns a List of Items

```
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());public
    }
    // Return the items
    return items;
}
```

You can then access the list from the one component on the page, as shown in Example 12–8.

Example 12–8 Accessing the Items from a JSF Page

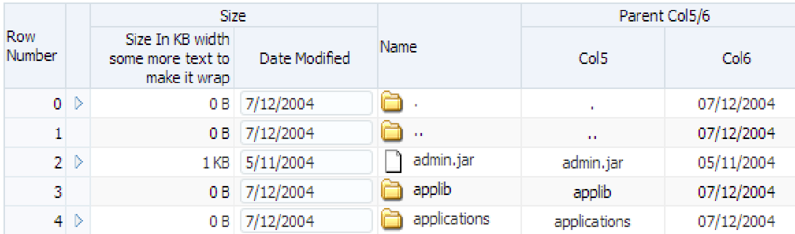
```
<af:table var="row">
  <af:column>
    <af:selectOneChoice value="#{row.myValue}">
      <f:Items value="#{page_backing.Items}"/>
    </af:selectOneChoice>
  </af:column>
</af:table>
```

12.4 Adding Hidden Capabilities to a Table

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.

Note: When a table that uses the `detailStamp` facet is rendered in Screen Reader mode, the contents of the facet appear in a popup window. For more information about accessibility, see Chapter 33, "Developing Accessible ADF Faces Pages."

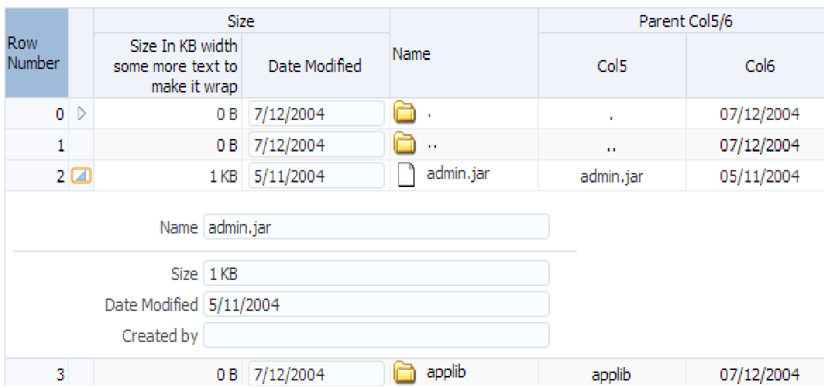
Figure 12–23 Table with Unexpanded DetailStamp Facet



Row Number		Size: Size In KB width some more text to make it wrap	Size: Date Modified	Name	Parent Col5/6: Col5	Parent Col5/6: 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.

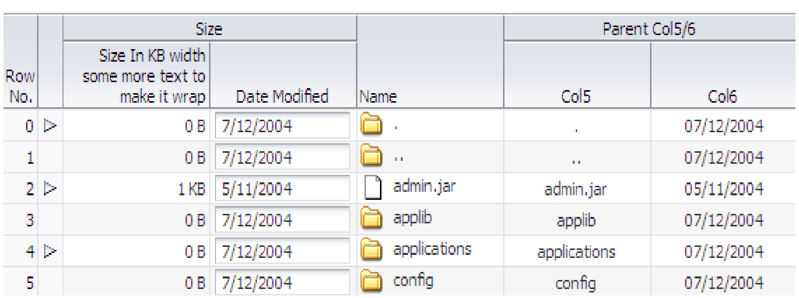
Figure 12–24 Expanded detailStamp Facet



Row Number		Size: Size In KB width some more text to make it wrap	Size: Date Modified	Name	Parent Col5/6: Col5	Parent Col5/6: 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
Name: admin.jar Size: 1 KB Date Modified: 5/11/2004 Created by:						
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: Size In KB width some more text to make it wrap	Size: Date Modified	Name	Parent Col5/6: Col5	Parent Col5/6: 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.

12.4.1 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:

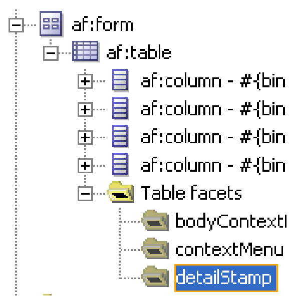
It may be helpful to have an understanding of how the attributes can affect functionality. For more information, see Section 12.4, "Adding Hidden Capabilities to a Table."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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.

   Example 12–9 shows abbreviated code used to display the `detailStamp` facet shown in Figure 12–24, which shows details about the selected row.

Example 12–9 Code for detailStamp Facet

```
<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.

12.4.2 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.

12.5 Enabling Filtering in Tables

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

			1700
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. For more information see the "Creating ADF Databound Tables" chapter of *Developing Web User Interfaces with Oracle ADF Faces*

In Example 12–10, the table `filterVisible` attribute is set to `true` to enable the filter input fields, and the `sortProperty` attribute is set on the column to identify the column in the `filterModel` instance. Each column element has its `filterable` attribute set to `true`.

Example 12–10 Table Component with Filtering Enabled

```
<af:table value="#{myBean.products}" var="row"
          queryListener="#{tableFilter.processTableFilter}"
          filterModel="#{tableFilter.queryDescriptor}"
               ...
          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>
```

12.5.1 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 columns that will use filtering must either have a value for the `headerText` attribute, or must contain a component in the `header` facet. 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. For more information, see Section 12.5, "Enabling Filtering in Tables."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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. Your implementation must return a `ConjunctionCriterion` object. For more information about this class, see the *Java API Reference for Oracle ADF Faces*.
2. Create a table, as described in Section 12.3, "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.
   - **QueryListener**: Bind to a listener method that will process the table filter.

   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:

   1. In the Structure window, right-click the column to be filtered and choose **Insert inside af:column** > **JSF Core** > **Filter facet**.
   2. From the Components window, drag and drop a component into the facet.
   3. Set the value of the component to be the `ConjunctionCriterion` object returned from the `getFilterConjunctionCriterion` method in `FilterableQueryDescriptor` class created in Step 1. Note that the value must take into account the variable used for the row, for example:

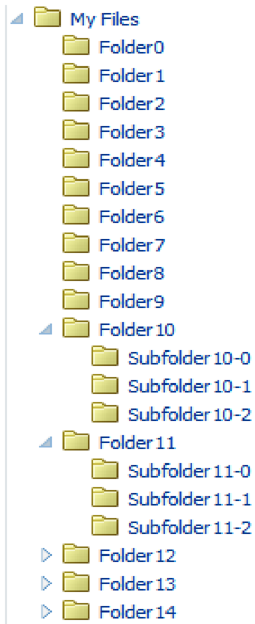
   ```
   #{af:inputDate label="Select Date" id="name"
                  value="row.queryDescriptor.
                         filterConjunctionCriterion.criterionMap.
                          Hiredate.value"}
   ```
4. In the Structure window, select a column in the table and in the Properties window, 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.

12.6 Displaying Data in Trees

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 Section 12.3, "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 Example 12–11.

Example 12–11 Constructing a TreeModel

```
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 _nodeVal(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. For more information see the "Displaying Master-Detail Data" chapter of *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`. For more information, see Section 12.6.4, "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 Example 12–12, 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:

Example 12–12 Displaying Data in a Tree

```
<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 more information about using the `panelCollection` component, see Section 12.9, "Displaying Table Menus, Toolbars, and Status Bars."

12.6.1 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. For more information, see Section 12.6, "Displaying Data in Trees."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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, as shown in Example 12–11.
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 programatically. 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`), or you want the user to click a node to make it editable (`clickToEdit`). For more information, see Section 12.2.4, "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 more information about context menus, see Chapter 15, "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 programatically perform some action on the selected nodes, see Section 12.6.5, "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. For more information, see Section 12.2.2, "Content Delivery."

- **FetchSize**: Specify the number of rows in the data fetch block. For more information, see Section 12.2.2, "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.

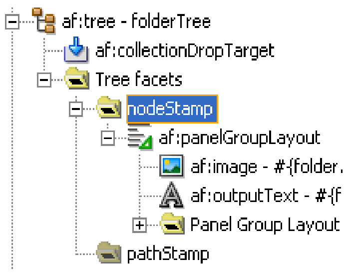
Note: Note the following about setting the `autoHeightRows` attribute:

- 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). For more information, see Section 12.2.7, "Geometry Management for the Table, Tree, and Tree Table Components."

- **SelectionListener**: Optionally enter an EL expression for a listener that handles selection events. For more information, see Section 12.6.5, "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. For more information, see Section 12.6.4, "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. For more information, see Section 12.6.5, "What You May Need to Know About Programmatically Selecting Nodes."
8. 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).

12.6.2 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. Example 12–13 shows the abbreviated code for the tree in the File Explorer application that displays the directory structure.

Example 12–13 ADF Faces Tree Code in a JSF Page

```
<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:outputText id="folderNodeStampText" value="#{folder.name}"/>
    </af:panelGroupLayout>
  </f:facet>
</af:tree>
```

12.6.3 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. For more information, see Section 12.6.4, "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. For more information, see Section 12.6.5, "What You May Need to Know About Programmatically Selecting Nodes."

12.6.4 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 Example 12–14.

Example 12–14 Tree Table Component with rowDisclosureListener

```
<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}">
```

The backing bean method that handles row disclosure events is shown in Example 12–15. The example illustrates expansion of a tree node. For the contraction of a tree node, you would use `getRemovedSet`.

Example 12–15 Backing Bean Method for RowDisclosureEvent

```
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 an node in the hierarchy programatically, as shown in Example 12–16. 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 and `removeAll()` method removes all the nodes from the set.

Example 12–16 Tree Component with disclosedRowKeys Attribute

```
<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 Example 12–17.

Example 12–17 Backing Bean Method for Handling Row Keys

```
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;
}
```

12.6.5 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.

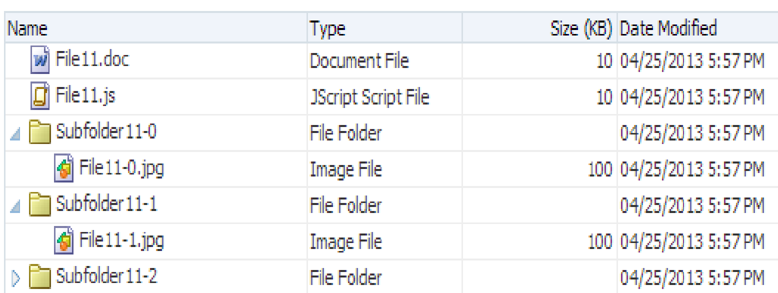
When a user selects or deselects a node, the tree component fires a `selectionEvent` event. This event has two `RowKeySet` objects: the `RemovedSet` object for all the deselected nodes and the `AddedSet` object for all the selected nodes.

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 programatically. 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 and `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 Section 12.3.8, "What You May Need to Know About Performing an Action on Selected Rows in Tables."

12.7 Displaying Data in Tree Tables

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



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 an node in the hierarchy. The `treeTable` component supports the same stamping behavior as the `Tree` component (for details, see Section 12.6, "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. Example 12–18 shows abbreviated code for the tree table in the File Explorer application.

Example 12–18 Stamping Rows in a TreeTable

```
<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>
```

The tree table component supports many of the same attributes as both tables and trees. For more information about these attributes see Section 12.3, "Displaying Data in Tables" and Section 12.6, "Displaying Data in Trees."

12.7.1 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. For more information, see Section 12.7, "Displaying Data in Tree Tables."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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 Section 12.3.4, "How to Display a Table on a Page," and Section 12.6.1, "How to Display Data in Trees" for help in configuring the attributes.

12.8 Passing a Row as a Value

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 Section 5.7, "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 Example 12–19. 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 outcome `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.

Example 12–19 Using SetPropertyListener and PageFlowScope

```
<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 Example 12–20.

Example 12–20 Retrieving PageFlowScope Objects

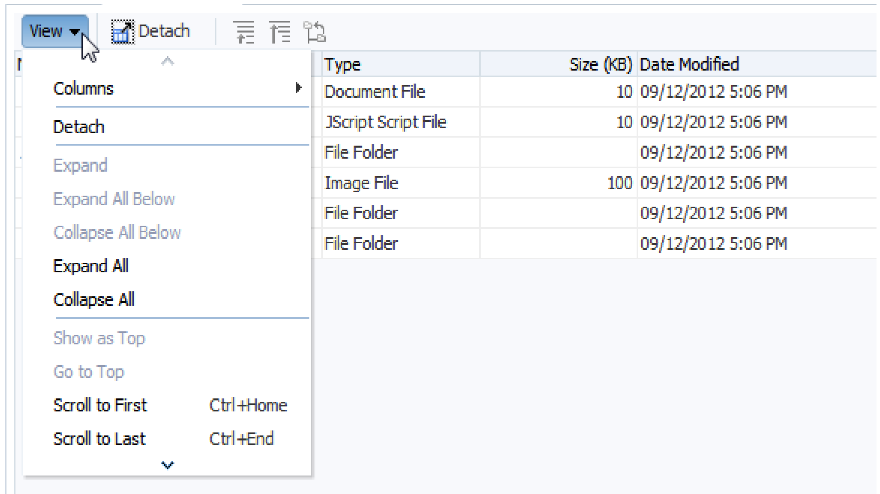
```
<h:panelGrid columns="2">
  <af:outputText value="Firstname:"/>
  <af:inputText value="#{pageFlowScope.empDetail.name}"/>
  <af:outputText value="Email:"/>
  <af:inputText value="#{pageFlowScope.empDetail.email}"/>
  <af:outputText value="Hiredate:"/>
  <af:inputText value="#{pageFlowScope.empDetail.hiredate}"/>
  <af:outputText value="Salary:"/>
  <af:inputText value="#{pageFlowScope.empDetail.salary}"/>
</h:panelGrid>
```

12.9 Displaying Table Menus, Toolbars, and Status Bars

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 more information about menus, toolbars, and toolbar buttons, see Chapter 16, "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.

Example 12–21 shows how the `panelCollection` component contains menus and toolbars.

Example 12–21 The panelCollection Component with Table, 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. For more information, see Section 16.2, "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.

12.9.1 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. For more information, see Section 12.9, "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. For more information, see Section 12.1.2, "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...
statusBar	status bar
viewMenu	View menu
formatMenu	Format menu
columnsMenuItem	Columns menu item in the View menu
columnsMenuItem:*colId* For example: `columnsMenuItem:col1, col2`	Columns with matching IDs in the Columns menu For example, the value to the left would not display the columns whose IDs are `col1` and `col2`
freezeMenuItem	Freeze menu item in the View menu
detachMenuItem	Detach menu item in the View menu
sortMenuItem	Sort menu item in the View menu
reorderColumnsMenuItem	Reorder Columns menu item in the View menu
resizeColumnsMenuItem	Resize Columns menu item in the Format menu
wrapMenuItem	Wrap menu item in the Format menu
showAsTopMenuItem	Show As Top menu item in the tree's View menu
scrollToFirstMenuItem	Scroll To First menu item in the tree's View menu
scrollToLastMenuItem	Scroll To Last menu item in the tree's View menu
freezeToolbarItem	Freeze toolbar item
detachToolbarItem	Detach toolbar item
wrapToolbarItem	Wrap toolbar item
showAsTopToolbarItem	Show As Top toolbar item
wrap	Wrap menu and toolbar items
freeze	Freeze menu and toolbar items
detach	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.

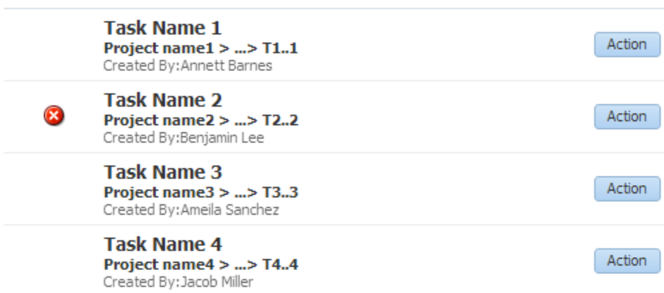
- View menu: Add `commandMenuItem` components to the `viewMenu` facet. For multiple items, use the `group` component as a container for the `commandMenuItem` components.

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 more instructions about menus and toolbars, see Chapter 16, "Using Menus, Toolbars, and Toolboxes."

12.10 Displaying a Collection in a List

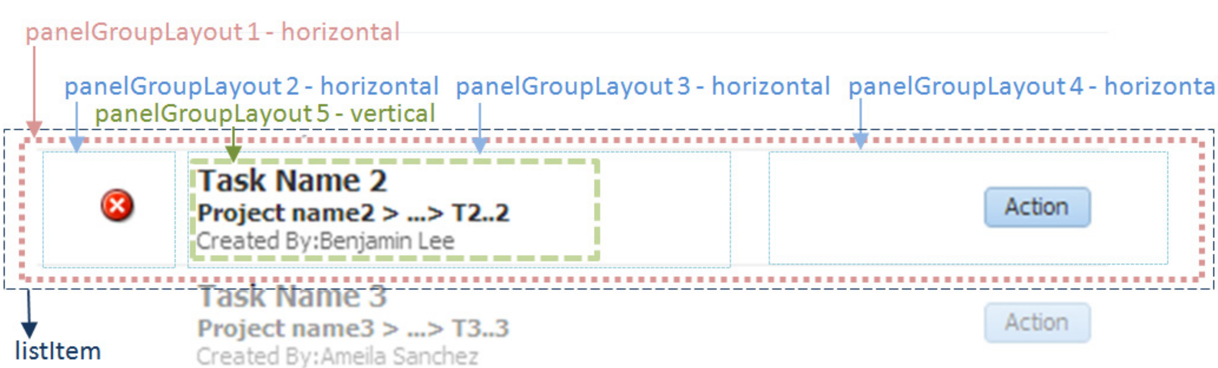
Instead of using a table with multiple columns, you can use the listView and listItem components to display structured data in a simple table-like format that contains just one column. Figure 12–32 shows a `listView` component that contains one `listItem` component used to display an error icon, task information, and an action button, for each row.

Figure 12–32 The listView Component with a listItem Component



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



Example 12–22 shows the corresponding code.

Example 12–22 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="i1"
                  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: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>
```

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.

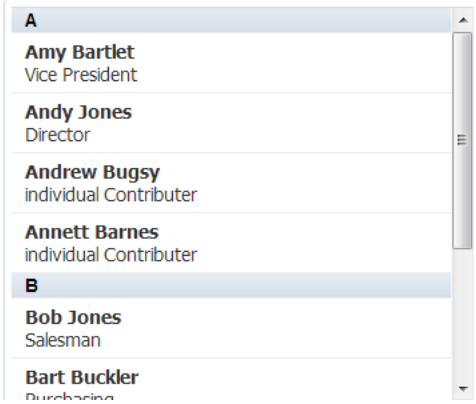
In this example, the `listView` value is bound to the `demolistView.taskModel` object. The properties on this object can be accessed using the `var` property, which is set to `item`. For example, in order to display the task name, the `outputText` component value is set to `item.taskName`.

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 more information about the TreeModel class, see Section 12.6, "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 Example 12–23.

Example 12–23 The groupHeaderStamp Facet in a listView Component

```
<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. For more information, see Section 12.6.4, "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. For more information, see Section 12.3.8, "What You May Need to Know About Performing an Action on Selected Rows in Tables."

12.10.1 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 more information about the `CollectionModel` class, see the MyFaces Trinidad Javadoc at `http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html`.

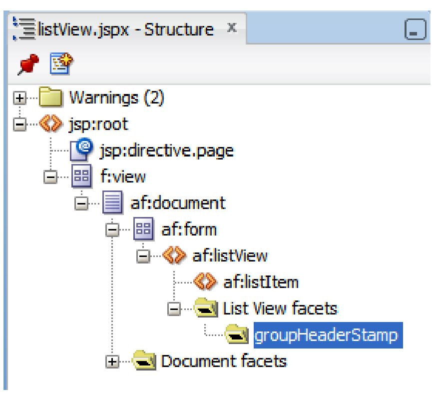
2. In the Components window, from the Data Views panel, drag and drop a **List View** on to the page.

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. For more information, see Section 12.6.5, "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 programatically perform some action on the selected rows, see Section 12.3.8, "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 12–22 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. Example 12–36 shows the `groupHeaderStamp` facet in the Structure window.

Figure 12–36 The groupHeaderStamp Facet in the Structure Window

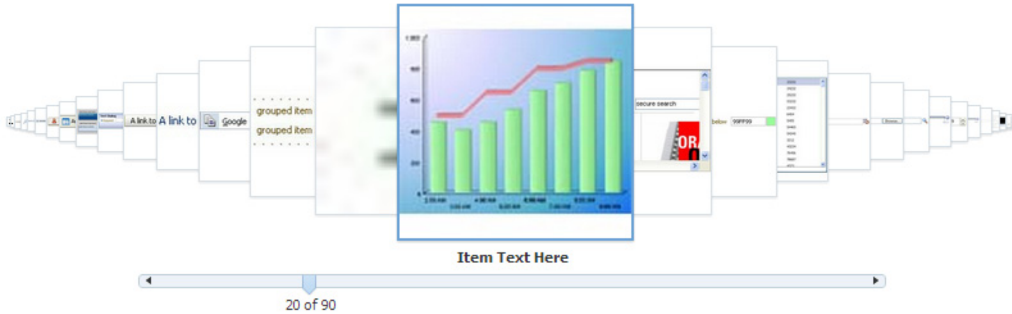


7. Drag and drop an **Output Text** into the `listItem` component to display your header text, and configure the `outputText` component as needed.

12.11 Displaying Images in a Carousel

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, and 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 or Fusion FX v1.2 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 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. Example 12–24 shows the JSF page code for the carousel. For more information about stamping behavior in a carousel, see Section 12.6, "Displaying Data in Trees."

Example 12–24 Carousel Component JSF Page Code

```
<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 more 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. For more information, 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.

12.11.1 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. For more information, see Section 12.11, "Displaying Images in a Carousel."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 12.1.2, "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 more 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 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. Example 12–25 shows the handler method on the `CarouselBean` which redraws the detail panel when the spin happens.

Example 12–25 Handler for the CarouselSpinEvent

```
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.

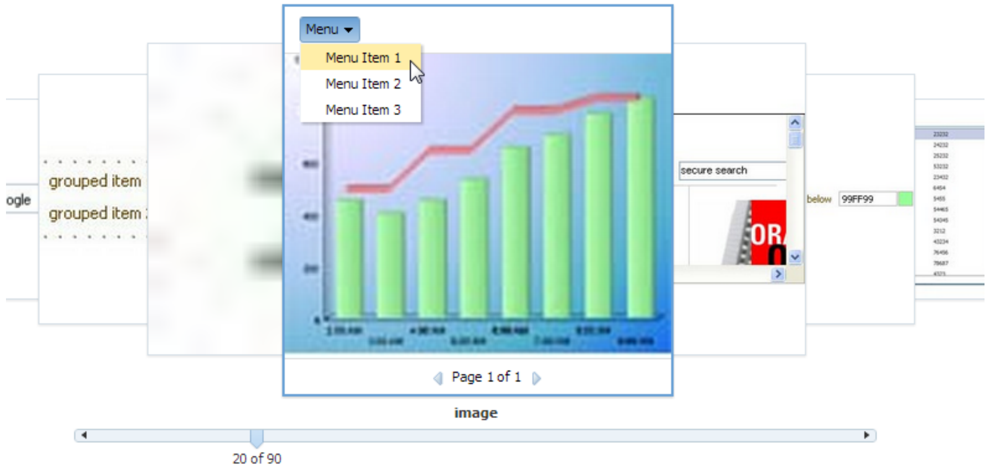
   Bind the `CarouselItem` component's attributes to the properties in the data model using the variable value set on the carousel's `var` attribute. For example, the carousel in Example 12–24 uses `item` as the value for the `var` attribute. So the value of the `carouselItem`'s `text` attribute would be `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, the carousel in Example 12–24 uses `item` as the value for the `var` attribute. So 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



Example 12–26 shows the corresponding page code.

Example 12–26 A More Complex Layout for a Carousel

```
<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"
                       halign="center">
    <af:panelGroupLayout id="footHorz" layout="horizontal">
      <af: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>
```

Example 12–27 shows the corresponding JavaScript.

Example 12–27 JavaScript Code to Handle Mouse Over and Mouse Down

```
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.PROPAGATE_LOCALLY);
          overlayFooterComponent.setProperty("inlineStyle", "display:block",
false, AdfUIComponent.PROPAGATE_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.

12.11.2 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

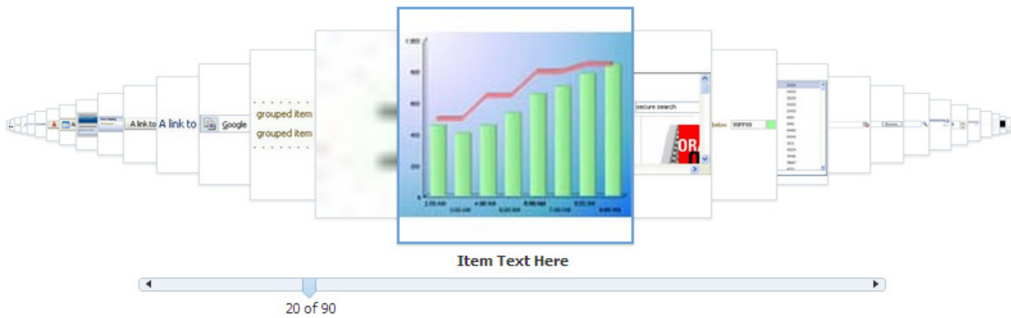
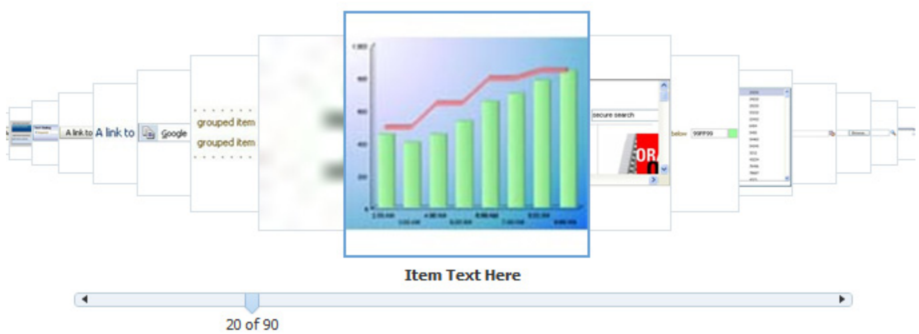


Figure 12–46 shows the same component in Internet Explorer.

Figure 12–46 Carousel Component Displayed in Microsoft Internet Explorer



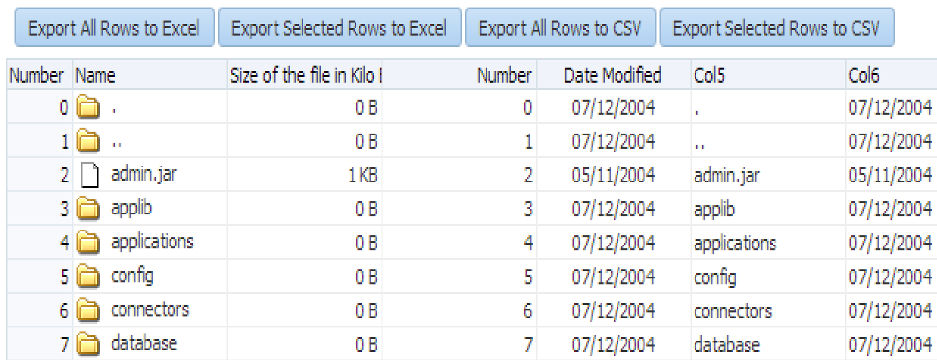
12.12 Exporting Data from Table, Tree, or Tree Table

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. 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. For more information, see Section 25.5.3, "Exporting from a Pivot Table."

For example, Figure 12–47 shows the table from the ADF Faces 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



Number	Name	Size of the file in Kilo I	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.

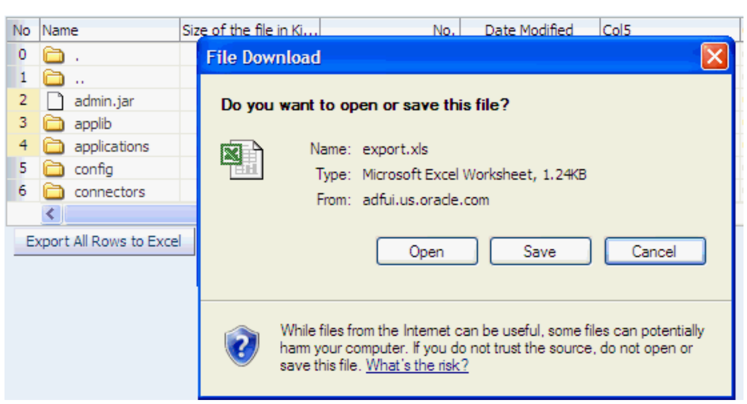
Only the following can be exported:

- Value of value holder components (such as `input` and `output` components).
- Value of `selectItem` components used in `selelctOneChoice` 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.

12.12.1 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. For more information, see Section 12.12, "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. For more information, see Section 12.1.2, "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 Section 12.3, "Displaying Data in Tables."

> **Tip:** If you want users to be able to select rows to export, then configure your table to allow selection. For more information, see Section 12.3.2, "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. For more information, see Section 12.9, "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 an 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 more information about the `detailStamp` facet, see Section 12.4, "Adding Hidden Capabilities to a Table"). Set to one of the following:
     - `all`: All rows will be automatically selected and exported.
     - `selected`: Only the rows the user has selected will be exported.
     - `allWithoutDetails`: All rows, except the data in the `detailStamp` facet, will be selected and exported.
     - `selectedWithoutDetails`: Only the rows the user has selected will be exported, except for the data in the `detailStamp` facet.
   - **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.

Example 12–28 shows the code for a table and its `exportCollectionActionListener` component. Note that the `exportedId` value is set to the table `id` value.

Example 12–28 Using the exportCollectionActionListener to Export a Table

```
<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"/>
```

12.12.2 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.

12.13 Accessing Selected Values on the Client from Collection-Based Components

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.

12.13.1 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. For more information, see Section 12.1.2, "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 Example 12–29 the table uses an `outputText` component to display the stamped rows.

Example 12–29 Table Component Uses an outputText Component for 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 Example 12–30.

Example 12–30 Using a clientListener to Register a Selection

```
<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 Example 12–31 extracts the first row key (which in this case, is the only row key).

Example 12–31 Iterating Over Row Keys Using a "for" in Loop

```
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. Example 12–32 shows this, where `nameStamp` is the ID of the table.

Example 12–32 Finding a Stamped Component Instance Given a Selected Row

```
// 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, Example 12–33 shows how to use the stamped component to get the row-specific value of the `name` attribute (which was the stamped value as shown in Example 12–29)and then display the name in an alert.

Example 12–33 Retrieving the Name of the Row in a Stamped Component

```
if (nameStamp)
  {
    // This is the row-specific name
    var name = nameStamp.getValue();

    alert("The selected name is: " + name);
  }
```

Example 12–34 shows the entire code for the JavaScript.

Example 12–34 JavaScript Used to Access Selected Row Value

```
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);
  }
}
```

12.13.2 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:

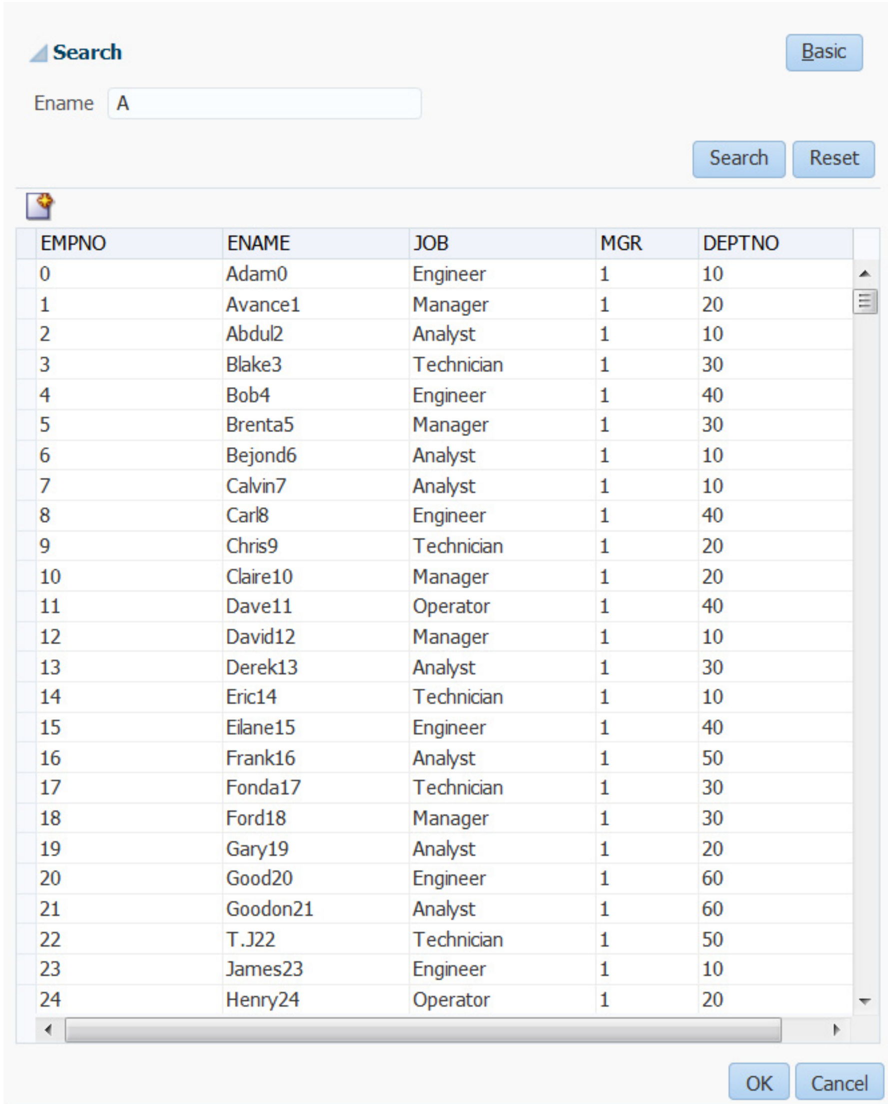
13.1 About List-of-Values Components

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

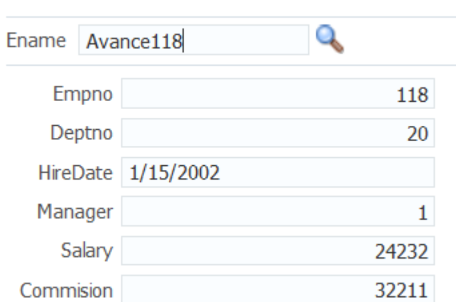
In this form, the employee name field is an LOV that contains a list of employees. When the user clicks the search icon of the `inputListOfValues` component, a Search and Select popup dialog displays all employees, along with a search field that allows the user to search for the employee, as shown in 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



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



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. Example 13–1 shows the code to display custom content using a table component.

Example 13–1 Adding Custom Content to the Search and Select Dialog

```
<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, and have the dialog populated with the rows that match the partial criteria. For this to work, you must implement logic so that when the user tabs 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.

If you want to add the auto-complete feature when the user tabs 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 out. Clicking on the Search link will still launch the Search and Select dialog. Example 13–2 shows the listener code in a managed bean that is used to disable the custom popup.

Example 13–2 Disabling the Custom Popup

```
public void LaunchPopupListener(LaunchPopupEvent launchPopupEvent) {
    if (launchPopupEvent.getPopupType().equals
       (LaunchPopupEvent.PopupType.SEARCH_DIALOG)
    {
      ...

      launchPopupEvent.setLaunchPopup(false);
    }
}
```

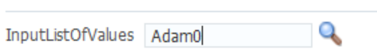
If the `readOnly` attribute is set to `true`, the input field is disabled. If `readOnly` is set to `false`, then the `editMode` attribute determines which type of input is allowed. If `editMode` is set to `select`, the value can be entered only by selecting from the list. If `editMode` is set to `input`, 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. Example 13–3 shows the code for an LOV component with both auto-suggest behavior and a smart list.

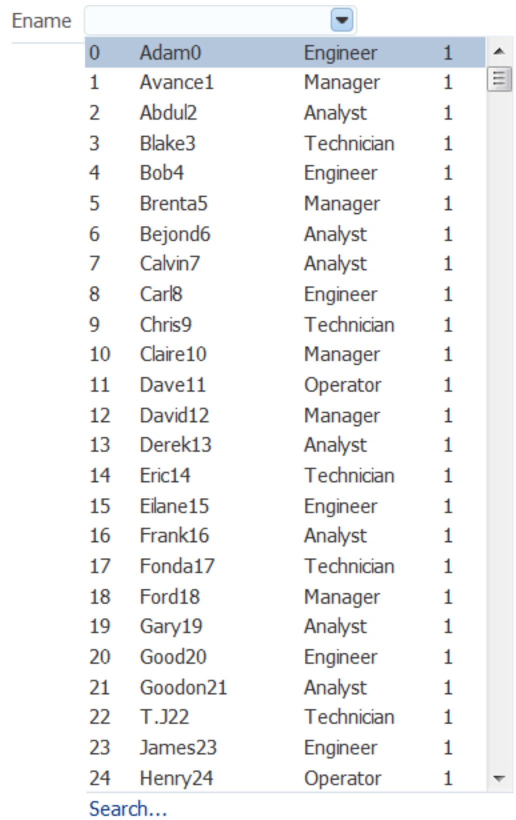
Example 13–3 Auto-Suggest Behavior and Smart List

```
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



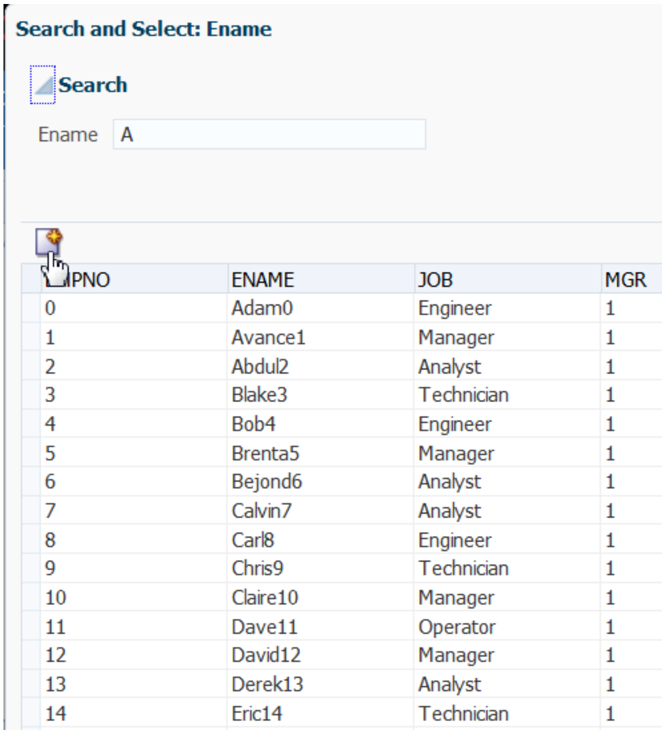
The dropdown list of the `inputComboboxListOfValues` component can display the following:

- 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. For more information, see the "Creating an Input Table" section 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. For more information, see the "Creating Databound Selection Lists and Shuttles" chapter of *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 Chapter 14, "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.

13.1.1 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 Chapter 4, "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. For more information, 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. For more information, see Chapter 6, "Handling Events."
- You can add validation and conversion to list-of-values components. For more information, see Chapter 7, "Validating and Converting Input."
- You can display tips and messages, as well as associate online help with list-of-values components. For more information, see Chapter 19, "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. For more information, see Section 5.2, "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. For more information, see Section 5.3, "Using the Optimized Lifecycle."
- You can change the appearance of the components using skins. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."
- You can make your list-of-values components accessible. For more information, see Chapter 33, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."
- The LOV components use the query component to populate the search list. For more information on the query component, see Chapter 14, "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 more information about select choice components, list box components, and radio buttons, see Chapter 11, "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). For more information, see the "Creating a Basic Databound Page" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

13.2 Creating the ListOfValues Data Model

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 more information on the LOV data model, see the *Java API Reference for Oracle ADF Faces*.

13.2.1 How to Create the ListOfValues Data Model

Begin you begin:

It may be helpful to have an understanding of the list-of-values data model. For more information, see Section 13.2, "Creating the ListOfValues Data Model."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 13.1.1, "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
`autoCompleteValue()`	Called when the search icon is clicked or the value is changed and the user tabs out from the input field, as long as `autoSubmit` is set to `true` 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.

Table 13–1 (Cont.) ListOfValues Model API

Method	Functionality
`valueSelected(value)`	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.
`isAutoCompleteEnabled()`	Returns a boolean to decide whether or not the auto complete is enabled.
`getTableModel()`	Returns the implementation of the `TableModel` class, on which the table in the search and select dialog will be based and created.
`getItems()` and `getRecentItems()`	Return the `items` and `recentItems` lists to be displayed in the combobox dropdown. Valid only for the `inputComboboxListOfValues` component. Returns null for the `inputListOfValues` component.
`getItemDescriptors()`	Return the list of `columnDescriptors` to be displayed in the dropdown list for an `inputComboboxListOfValues` component.
`getQueryModel()` and `getQueryDescriptor()`	Return the `queryModel` based on which the `query` component inside the Search and Select dialog is created.
`performQuery()`	Called when the search button in the `query` component is clicked.

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.

2. 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.

3. 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.

13.3 Using the inputListOfValues Component

The `inputListOfValues` component uses the `ListOfValues` model you implemented to access the list of items, as documented in Section 13.2, "Creating the ListOfValues Data Model."

13.3.1 How to Use the InputListOfValues Component

Before you begin:

It may be helpful to have an understanding of the `inputListOfValues` component. For more information, see Section 13.3, "Using the inputListOfValues Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 13.1.1, "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 Section 13.2.1, "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 Section 13.2.1, "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.

   The rest of the attributes in this section can be populated in the same manner as with any other input component. For more information, see Section 11.3, "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. For more information, see Section 11.3, "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. For more information, see Chapter 15, "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.
   - 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 Example 13–4

Example 13–4 autoSuggestBehavior Tag in an LOV

```
<af:inputListOfValues value="#{bean.value}" id="inputId">
    ...
    <af:autoSuggestBehavior
          suggestItems="#{bean.suggestItems}"
          smartList="#{bean.smartList}"
          maxSuggestedItems="7"/>
</af:inputListOfValues>
```

   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 Example 13–5.

Example 13–5 suggestItems Method Signature

```
List<javax.model.SelectItem> suggestItems(javax.faces.context.FacesContext,
    oracle.adf.view.rich.model.AutoSuggestUIHints)
```

13.3.2 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 Example 13–6. The default value of the `-tr-stretch-search-dialog` selector key is `true`. For more information about skinning, see the skinning chapter.

Example 13–6 Disabling the Resizing of Search and Select Dialogs

```
af|inputComboboxListOfValues{
  -tr-stretch-search-dialog: false;
}
af|inputListOfValues{
  -tr-stretch-search-dialog: false;
}
```

13.4 Using the InputComboboxListOfValues Component

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 Section 13.2, "Creating the ListOfValues Data Model."

For more information about skinning and the Search and Select dialog sizing, see Section 13.3.2, "What You May Need to Know About Skinning the Search and Select Dialogs in the LOV Components."

13.4.1 How to Use the InputComboboxListOfValues Component

Before you begin:

It may be helpful to have an understanding of the `inputComboboxListOfValues` component. For more information, see Section 13.4, "Using the InputComboboxListOfValues Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 13.1.1, "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 Section 13.2.1, "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 Section 13.2.1, "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.

   The rest of the attributes in this section can be populated in the same manner as with any other input component. For more information, see Section 11.3, "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. For more information, see Section 11.3, "Using the inputText Component."

5. 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.

6. If you want users to be able to create a new item, create a popup dialog with the ID given in Step 5. For more information, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

7. 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.

8. 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 Example 13–7.

Example 13–7 autoSuggestBehavior Tag in an LOV

```
<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.

9. If you are not using the component with ADF Model, create the `suggestItems` method to process and display the list. The `suggestItems` method signature is shown in Example 13–8.

Example 13–8 suggestItems Method Signature

```
List<javax.model.SelectItem> suggestItems(javax.faces.context.FacesContext,
    oracle.adf.view.rich.model.AutoSuggestUIHints)
```

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:

14.1 About Query Components

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:

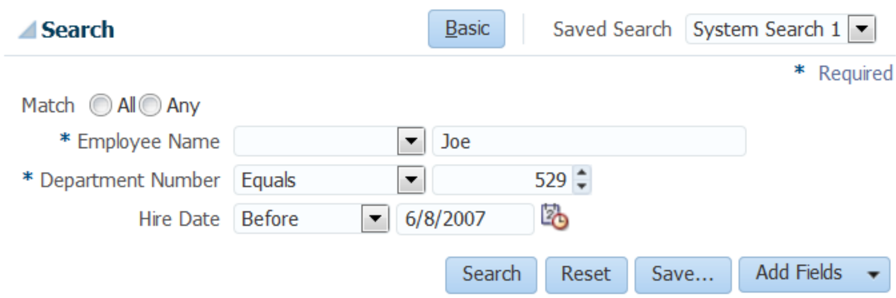
- Selecting and searching against multiple search criteria
- Dynamically adding and deleting criteria items
- Selecting search operators (associated to a single criterion)
- Choosing match all or match any conjunction
- Displaying in a basic or advanced mode
- Creating saved searches
- Personalizing 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 Chapter 35, "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. For more information, see the "Creating ADF Databound Search Forms" chapter of *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 more 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 more 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
- 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.

14.1.1 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.

14.1.2 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. For more information, see the *JavaScript API Reference for Oracle ADF Faces*.
- You can display tips and messages, as well as associate online help with query components. For more information, see Chapter 19, "Displaying Tips, Messages, and Help."
- You can change appearance of the components using skins. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."
- You can make your query components accessible. For more information, see Chapter 33, "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. For more information, see Chapter 32, "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). For more information, see the "Creating Databound Search Forms" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

14.2 Creating the Query Data Model

Before you can use the `query` components, you must to create your `QueryModel` classes. For more 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. For more information, see Section 14.3.2, "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

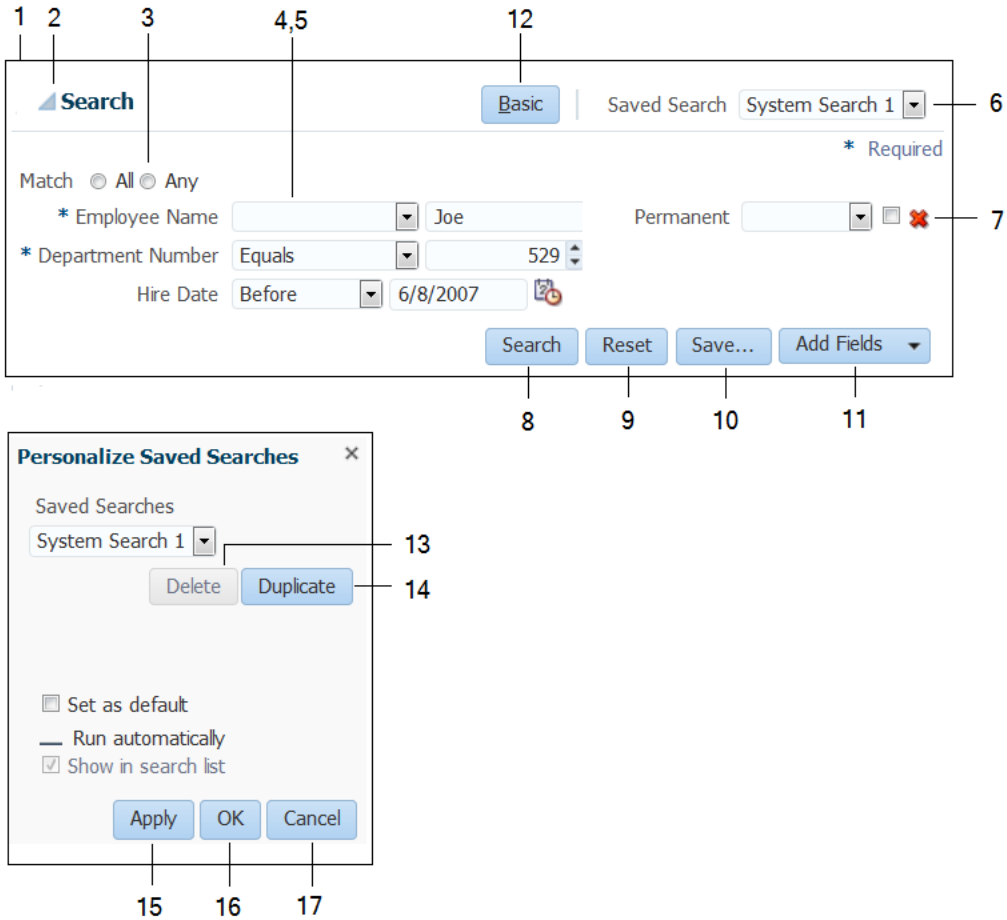


Table 14–1 shows UI artifacts rendered for the `query` component, the associated class, class property, and methods used by the artifact.

Table 14–1 Query UI Artifacts and Associated Model Class Operations and Properties

	UI Artifact	Class Property/Methods Used	Comments
1	Search panel	The `QueryDescriptor` 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 `getConjunction()` method on the `ConjunctionCriterion` class.	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. For example, if the `All` conjunction type is used between all the search fields, then `All` appears selected. If it is a mix of `All` and `Any`, then none of the radio buttons appears selected. The Match Type will be read only if the `conjunctionReadOnly` property is set to `true`. Its not rendered at all when the `displayMode` attribute is set to `simple`.

Table 14–1 (Cont.) Query UI Artifacts and Associated Model Class Operations and

	UI Artifact	Class Property/Methods Used	Comments
4	Group of search fields	The collection of search fields for a `QueryDescriptor` object is represented by a `ConjunctionCriterion` object, returned by the method `getConjunctionCriterion()` on the `QueryDescriptor` class. The `getCriterionList()` method returns a `List<Criterion>` list.	Displays one or more search fields associated with the currently selected search.
5	Search field	An `AttributeCriterion` class provides information specific to a search field instance. An `AttributeCriterion` object is an item in the `List<Criterion>` list returned by `getCriterionList()` method on the `ConjunctionCriterion` class (see #4). An `AttributeDescriptor` class provides static information pertaining to a search field. This is available through the method `getAttribute()`, on the `AttributeCriterion` class. The `getConverter()` method of the `AttributeDescriptor` class can be overridden to return a converter object of type `javax.faces.convert.Converter`. When defined, the attribute value is converted using this converter instance. The default return value is `null`. The `hasDependentCriterion` method in the `AttributeCriterion` class returns `true` 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 `false`.	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. For example, assume an Employee business object contains the search fields Employee **Name** and **Salary**. A user can then configure two different searches: one named Low Salaried Employees and one named High Salaried Employees. Both searches contain two search fields based on the `Employee` and `Salary` attributes. Even though both saved searches are based on the same attributes of the Employee object, the search field **Salary** is configured to have its default operator as **less than** and value as **50000.00** for the low Salaried Employees search and for the High Salaried Employees search, with a default operator of **greater than** and value of **100000.00**. Selecting the saved searches on the UI will show the appropriate operator and values for that search. Regardless of the search selected by the user, the search field for **Salary** always has to render a number component, and the label always has to show **Salary**.
6	Saved Searches dropdown	System- and user-saved searches are available through the methods `getSystemQueries()` and `getUserQueries()` on the `QueryModel` class.	Displays a list of available system- and user-saved searches. Saved searches are listed in alphabetical order. A **Personalize** option is also added if the `saveQueryMode` 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.

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

	UI Artifact	Class Method Invoked	Event Generated	Comments
7	Delete icon	During the Invoke Application phase, the method `removeCriterion()` on the `QueryDescriptor` class is called automatically by an internal `ActionListener` handler registered with the command component.	`ActionEvent`	Deletes a search field from the current `QueryDescriptor` object.
8	Search button	During the Apply Request Values phase of the JSF lifecycle, a `QueryEvent` event is queued, to be broadcast during the Invoke Application phase. 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 (for more information, see Section 14.4.1, "How to Add the Query Component"). These expressions should invoke the `get/setOperator();` `get/setOperators();` and `getValues()` methods, respectively, on the `AttributeCriterion` class. During the Invoke Application phase, the `QueryListener` registered with the `query` component is invoked and this performs the search. You must implement this listener.	`QueryEvent`	Rendered always on the footer (footer contents are not rendered at all when the `displayMode` attribute is `simple`) Performs a query using the select operator and selected Match radio (if no selection is made the default is used), and the values entered for every search field.
9	Reset button	During the Apply Request Values phase of the JSF lifecycle, a `QueryOperationEvent` event is queued with the operation type `QueryOperationEvent.Operation.RESET`, to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method `reset()` on the `QueryModel` class is called. This is done automatically by an internal `QueryOperationListener` handler registered with the `query` component. You must override this method to reset the `QueryDescriptor` object to its original state.	`QueryOperationEvent` (an internal `QueryOperation Listener` handler is registered with the `query` component that in turn calls the model methods).	Resets the search fields to its previous saved state.

Table 14–2 (Cont.) UI Artifact Behaviors and Associated Methods

	UI Artifact	Class Method Invoked	Event Generated	Comments
10	Save button	During the Apply Request Values phase of the JSF lifecycle, a `QueryOperationEvent` event is queued with the operation type `QueryOperationEvent.Operation.SAVE`, to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method `create()` on the `QueryModel` class is called. After the call to the `create()` method, the `update()` 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 `QueryOperationListener` handler registered with the `query` component. You must override this method to create a new object based on the argument passed in.	`QueryOperationEvent` (an internal `QueryOperationListener` 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.
11	Add Fields dropdown list	During the Invoke Application phase, the method `addCriterion()` on the `QueryDescriptor` class is called automatically by an internal `ActionListener` handler registered with the command component. You must override this method to create a new `AttributeCriterion` object based on the `AttributeDescriptor` object (identified by the name argument).	`ActionEvent`	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 `QueryOperationEvent` event is queued with the operation type `QueryOperationEvent.Operation.MODE_CHANGE`, to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method changeMode()on the `QueryModel` class is called.	`QueryOperationEvent` (an internal `QueryOperationListener` 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

	UI Artifact	Class Method Invoked	Event Generated	Comments
13	Delete button	During the Invoke Application phase, the method `delete()` on the `QueryModel` class is called. This is done automatically by an internal `QueryOperationListener` handler registered with the `query` component. You must override this method order to delete the `QueryDescriptor` object.	`ActionEvent`	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 `QueryOperationEvent` event is queued with the operation type `QueryOperationEvent.Operation.DUPLICATE`, to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method `update()` on the `QueryModel` class is called. This is done automatically by an internal `QueryOperationListener` handler registered with the `query` component. You must override this method in order to update the `QueryDescriptor` object using the arguments passed in.	`QueryOperationEvent` (an internal `QueryOperationListener` 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

	UI Artifact	Class Method Invoked	Event Generated	Comments
15	Apply button	During the Apply Request Values phase of the JSF lifecycle, a `QueryOperationEvent` event is queued with the operation type `QueryOperationEvent.Operation.UPDATE`, to be broadcast during the Invoke Application phase. During the Invoke Application phase, the method `update()` on the `QueryModel` class is called. This is done automatically by an internal `QueryOperationListener` handler registered with the `query` component. You must override this method in order to update the `QueryDescriptor` object using the arguments passed in.	`QueryOperationEvent` (an internal `QueryOperationListener` is registered with the `query` component that in turn calls the model methods).	Applies changes made to the selected saved search.
16	OK button	Same as the Apply button.	`QueryOperationEvent` (an internal `QueryOperationListener` handler is registered with the `query` component that in turn calls the model methods).	Applies changes made to the selected saved search and the dialog is closed afterwards.
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.

14.2.1 How to Create the Query Data Model

Begin you begin:

It may be helpful to have an understanding of the query data model. For more information, see Section 14.2, "Creating the Query Data Model."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 14.1.2, "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 lifecyle 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. Example 14–1 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.

Example 14–1 A QueryListener Handler Method

```
public void processQuery(QueryEvent event)
{
  DemoQueryDescriptor descriptor = (DemoQueryDescriptor) event.getDescriptor();
  String sqlString = descriptor.getSavedSearchDef().toString();
  setSqlString(sqlString);
}
```

14.3 Using the quickQuery Component

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. For more information, see Section 14.3.2, "How to Use a quickQuery Component Without a Model."

14.3.1 How to Add the quickQuery Component Using a Model

Before you begin

It may be helpful to have an understanding of forms and subforms. For more information, see Section 14.3, "Using the quickQuery Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 14.1.2, "Additional Functionality for the Query Components."

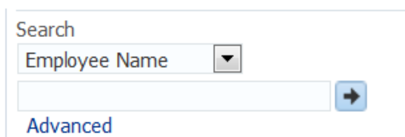
You will need to complete this task:

Create a `QueryModel` class and associated classes. For more information, see Section 14.2, "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 Section 14.2, "Creating the Query Data Model."
   - **value**: Enter an EL expression that evaluates to the class that implements the `QueryDescriptor` class, as created in Section 14.2, "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 Section 14.2, "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.

14.3.2 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 to 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. For more information, see Section 14.3, "Using the quickQuery Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 14.1.2, "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, form the Text and Selection panel, drag a **Select Item** onto the `selectOneChoice`. You can add as many as you need. For more information about using the `selectOneChoice` and `selectItems` components, see Section 11.6, "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.

14.3.3 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.

14.4 Using the query Component

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.

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	af:inputText	af:inputText
Ends with	af:inputText	af:inputText
Equals	Default list type hint	af:selectManyChoice
Does not equal	Default list type hint	af:selectManyChoice
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

Table 14–4 (Cont.) Rendered Component for Search Criteria Field of Type Number

Operator	Component	Component When Multiple Select Is Enabled
`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 more information about LOV, see Chapter 13, "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.

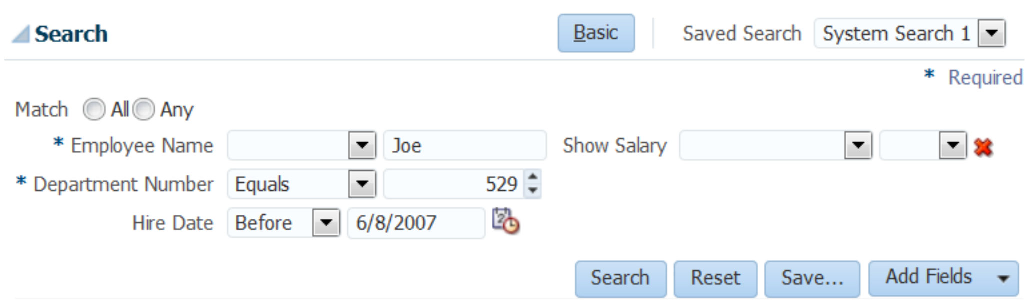
After the user enters all the search criteria values (including null values) and selects the **Match All** or **Match Any** radio button, the user can click the **Search** button to initiate the query. The query results can be displayed in any output component. Typically, the output component will be 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 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 more information about persistence, see Chapter 35, "Allowing User Customization on JSF Pages."

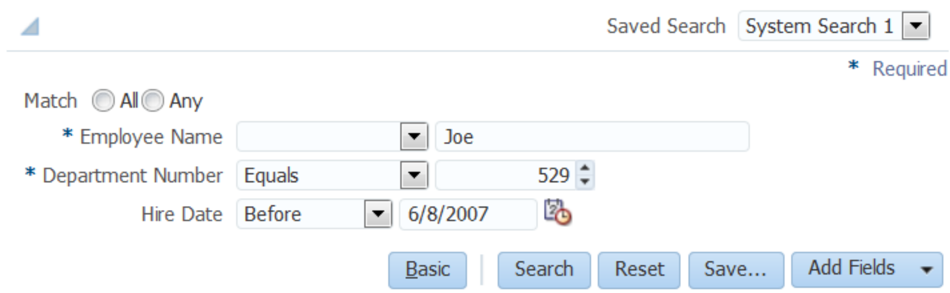
A seeded search is essentially a saved search that was created by the application developer. When the component is initialized, any seeded searches associated with that `query` component become available for the user to select.

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.

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 more information about persistence, see Chapter 35, "Allowing User Customization on JSF Pages."

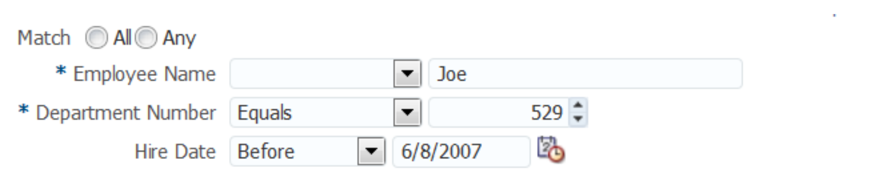
Along with the default display described previously, you can also configure the `query` component to display in a compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown list moves next to the expand or collapse icon. Figure 14–6 shows the same `query` 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.

14.4.1 How to Add the Query Component

Before you begin:

It may be helpful to have an understanding of forms and subforms. For more information, see Section 14.4, "Using the query Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 14.1.2, "Additional Functionality for the Query Components."

You will need to complete this task:

> Create a `QueryModel` class and associated classes. For more information, see Section 14.2, "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 Section 14.2, "Creating the Query Data Model."
   - **value**: Enter an EL expression that resolves to the `QueryDescriptor` class, as created in Section 14.2, "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.
   - **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 Section 14.2, "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 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:

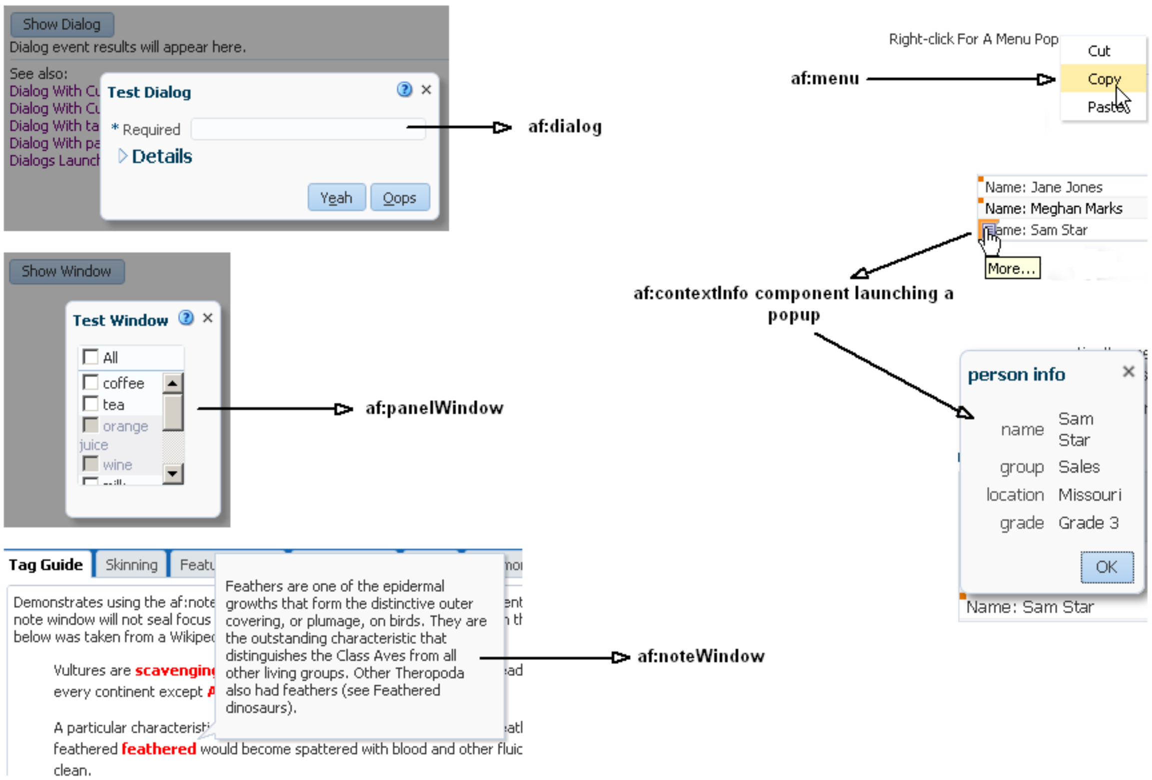
15.1 About Popup Dialogs, Menus, and Windows

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 15–1 shows examples where the `popup` component works with other ADF Faces components to render secondary windows.

Figure 15–1 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. For more information, see the "Running a Bounded Task Flow in a Modal Dialog" section 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. Example 15–1 shows how to set the value of the `LAST_WINDOW_SESSION_TIMEOUT` context parameter in a `web.xml` file to `1800` seconds.

Example 15–1 Specifying the Session Timeout for the Last Window in an Application

```
<!-- Sets the session timeout to 1800 seconds when there is only one window open
```

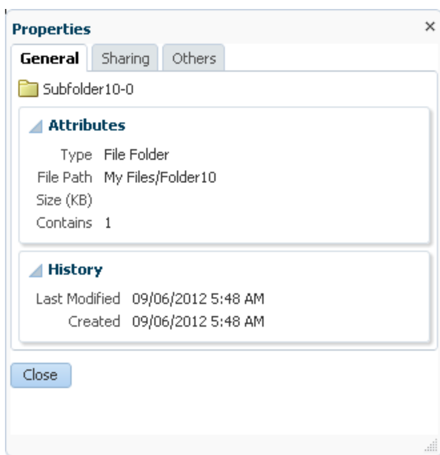
```
in the session and 1800 seconds is smaller then the original session timeout. This
gives your application the option to end the session when an end user closes the
last window. Specify a value in seconds. A negative value disables this feature.
The default value is -1. -->
   <context-param>
     <param-name>LAST_WINDOW_SESSION_TIMEOUT</param-name>
     <param-value>1800</param-value>
   </context-param>
```

For more information about configuring your application's `web.xml` file, Appendix A.2, "Configuration in web.xml."

15.1.1 Popup Dialogs, Menus, Windows Use Cases and Examples

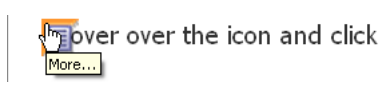
You can place a `dialog` component as a child to a `popup` component and render a dialog in a popup at runtime. The `dialog` component must be the only immediate child component of the `popup` component. At runtime, end users can view or enter information (for example, search criteria) and use the `dialog` component's default buttons to invoke a `dialogEvent` when clicked. Figure 15–2 shows an example where an end user can dismiss the dialog by clicking the **Close** button.

Figure 15–2 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 15–3.

Figure 15–3 With Mouseover, Larger Icon with Note Is Displayed



When the user clicks the note icon, the associated popup displays its enclosed content.

15.1.2 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. For more information, see Section 3.5.2, "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 more information about how to handle events on the server as well as on the client, see Chapter 6, "Handling Events."
- **Messages**: Popup dialogs and secondary windows are frequently used to provide different levels of help information for users. For more information about how to display messages to users, see Chapter 19, "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. For more information, see Chapter 32, "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. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."
- **Accessibility**: You can make your popup dialogs, menus, and windows accessible. For more information, see Chapter 33, "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. For more information, see the "Using Dialogs in Your Application" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

15.2 Declaratively Creating Popups

The `dialog`, `panelWindow`, `menu`, and `noteWindow` components can all be used inside the `popup` component to display inline popups, as shown in Table 15–1. When no child component exists for the `popup` component, a very simple inline popup appears.

Table 15–1 Components Used with the popup Component

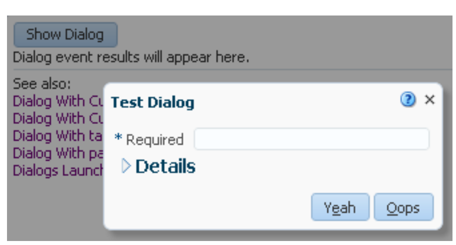
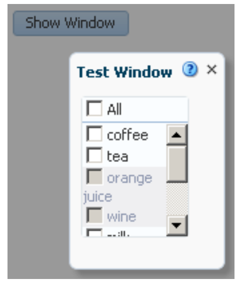
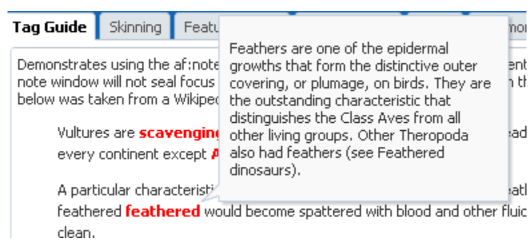
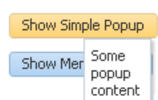
Component	Displays at Runtime
`dialog`	Displays its children inside a dialog and delivers events when the `OK`, `Yes`, `No`, and `Cancel` actions are activated. For more information, see Section 15.2.1, "How to Create a Dialog." 

Table 15–1 (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. For more information, see Section 15.2.2, "How to Create a Panel Window." 
`menu`	Displays a context menu for an associated component. For more information, see Section 15.2.3, "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. For more information, see Section 15.2.4, "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. For more information, see Chapter 19, "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. For more information, see Section 15.3, "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 Section 15.4, "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.

15.2.1 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 more information about the use of the af:popup component's autoCancel property, see Section 15.6, "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. For more information, see Section 15.2, "Declaratively Creating Popups."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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**: Select **true** to enable animation. Animation is determined by configuration in the trinidad-config.xml file and by its skin properties (for more information, see Section A.6.2.1, "Animation Enabled."). You can override this setting by selecting false.
   - **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. For more information, see Section 15.2.5, "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. For more information, see Section 15.6, "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`. For more information, see Section 15.2, "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 more information about using the `resetListener` component, see Section 15.7, "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.
   - **CloseIconVisible**: 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. For more information, see Section 9.2.1, "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 Example 15–2. 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.

Example 15–2 Handler for dialogEvent That Deletes an Item

```
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);
}
```

   Example 15–3 shows how the `dialogListener` attribute is bound to the `deleteItem` method.

Example 15–3 dialogListener Attribute Bound to a Method

```
<af:dialog title="#{explorerBundle['deletepopup.popuptitle']}"
           type="yesNo"
           dialogListener="#{explorer.headerManager.deleteItem}"
           id="d1">
```

   The `dialogEvent` is propagated to the server only when the outcome is `ok`, `yes`, or `no`. You can block this if needed. For more information, see Section 6.3.5, "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 Chapter 35, "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. For more information, see the "Running a Bounded Task Flow in a Modal Dialog" section in *Developing Fusion Web Applications with Oracle Application Development Framework*. For more information about the `showPopupBehavior` tag, see Section 15.3, "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. For more information, see Section 15.6, "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. For more information, see Section 15.3, "Declaratively Invoking a Popup."

15.2.2 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. For more information, see Section 15.2, "Declaratively Creating Popups."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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**: Select **true** to enable animation. Animation is determined by configuration in the `trinidad-config.xml` file and by its skin properties (for more information, see Section A.6.2.1, "Animation Enabled.").
- **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. For more information, see Section 15.2.5, "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. For more information, see Section 15.6, "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. For more information, see Section 9.2.1, "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 Chapter 35, "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 more information about the `showPopupBehavior` tag, see Section 15.3, "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. For more information, see Section 15.3, "Declaratively Invoking a Popup."

15.2.3 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 Section 16.3, "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. For more information, see Section 15.2, "Declaratively Creating Popups."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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**: Select **true** to enable animation. Animation is determined by configuration in the `trinidad-config.xml` file and by its skin properties (for more information, see Section A.6.2.1, "Animation Enabled."). You can override this setting by selecting `false`.

- **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. For more information, see Section 15.2.5, "What Happens at Runtime: Popup Component Events."
- Optionally, select a value from the **AutoCancel** dropdown list to determine the automatic cancel behavior. For more information, see Section 15.6, "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 Section 16.2.1, "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 Section 16.2.1, "How to Create and Use Menus in a Menu Bar."

4. Add logic on the parent page to invoke the popup and context menu. For more information, see Section 15.3, "Declaratively Invoking a Popup."

15.2.4 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. For more information, see Section 15.2, "Declaratively Creating Popups."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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**: Select **true** to enable animation. Animation is determined by configuration in the `trinidad-config.xml` file and by its skin properties (for

more information, see Section A.6.2.1, "Animation Enabled."). You can override this setting by selecting `false`.

- **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. For more information, see Section 15.2.5, "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. For more information, see Section 15.6, "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:
   1. Click the **Source** tab to view the page source code.
   2. Remove the closing slash (/) from the `af:noteWindow` tag.
   3. Below the `af:noteWindow tag`, enter the text to display, using simple HTML tags, and ending with a closed `af:noteWindow tag`.

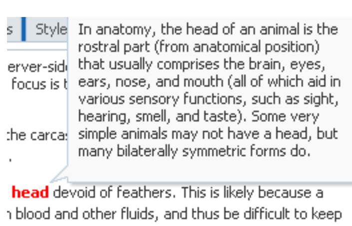
   Example 15–4 shows text for a note window.

Example 15–4 Text Within an af:noteWindow Tag

```
<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 15–4 shows how the note would display.

Figure 15–4 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. For more information, see Section 15.3, "Declaratively Invoking a Popup."

15.2.5 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 Example 15–5.

Example 15–5 Using eventContext for Shared Popups

```
<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

Example 15–6.

Example 15–6 Setting the Value of a Component in a Popup Using the launcherVar Property

```
<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: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 more information about using the `setPropertyListener` tag, see Section 5.7.2, "How to Use the pageFlowScope Scope Without Writing Java Code." For more information about using client attributes, see Section 4.6, "Using Bonus Attributes for Client-Side Components." For more information about the `showPopupBehavior` tag, see Section 15.3, "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. For more information, see Section 4.4, "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. For more information, see Section 6.4, "Sending Custom Events from the Client to the Server."

15.2.6 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 more information about the events raised by the `dialog` and `popup` components, see the *Tag Reference for Oracle ADF Faces*.

15.3 Declaratively Invoking a Popup

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 more information about client behavior tags, see Section 6.6, "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 Section 15.4, "Programmatically Invoking a Popup."

15.3.1 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. For more information, see Section 15.3, "Declaratively Invoking a Popup."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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 Section 15.2, "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 Section 6.4, "Sending Custom Events from the Client to the Server" or invoke the popup programmatically as documented in Section 15.4, "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.

15.3.2 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. Example 15–7 shows sample code that displays some text in the `af:popup` component with the `id` attribute "`popup2`" when the button "Show Popup" is clicked.

Example 15–7 showPopupBehavior Associated with Button Component

```
<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 Example 15–7 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 15–5.

Figure 15–5 Button and Popup Contents

15.4 Programmatically Invoking a Popup

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 Section 15.3, "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 Section 15.2, "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 Example 15–8.

Example 15–8 Binding a popup Component to a Backing Bean

```
<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. Example 15–9 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.

Example 15–9 Backing Bean Method Invoking a Popup

```
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 15–10 shows a backing bean method that cancels a popup in response to an actionEvent:

Example 15–10 Backing Bean Method Canceling a Popup

```
public void cancelPopupActionListener(ActionEvent event) {
   FacesContext context = FacesContext.getCurrentInstance();
   p1.cancel();
  }
```

Example 15–11 shows a backing bean method that hides a popup in response to an actionEvent:

Example 15–11 Backing Bean Method Hiding a Popup

```
public void hidePopupActionListener(ActionEvent event) {
   FacesContext context = FacesContext.getCurrentInstance();
   p1.hide();
 }
```

The p1 object in the previous examples refers to an instance of the RichPopup class from the following package:

oracle.adf.view.rich.component.rich.RichPopup

For more information about RichPopup, see the *Java API Reference for Oracle ADF Faces*.

15.4.1 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. For more information, see Section 15.4, "Programmatically Invoking a Popup."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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 Section 15.2, "Declaratively Creating Popups."

To programmatically invoke a popup:

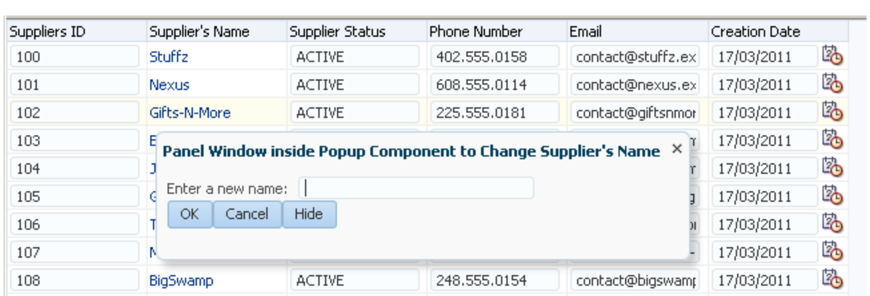
1. In the Components window, from the General Controls panel, drag and drop an action component onto the JSF page. For example, a **Button**.
2. In the Properties window, expand the **Behavior** section and set the following attributes:
   - **PartialSubmit**: Select **true** if you do not want the Fusion 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 more information about page rendering, see Chapter 8, "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`.

   For more information, see Example 15–9, "Backing Bean Method Invoking a Popup", Example 15–10, "Backing Bean Method Canceling a Popup", or Example 15–11, "Backing Bean Method Hiding a Popup".

15.4.2 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 15–6 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 15–6 Popup Component Invoked by a Server-Side Method

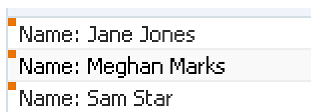


15.5 Displaying Contextual Information in Popups

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.

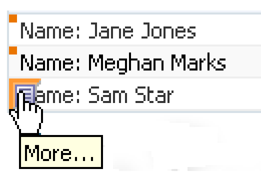
The `contextInfo` component allows you to display additional information in a popup and also notifies users that additional information is available. When you place the `contextInfo` component into the context facet of a component that supports contextual information, a small orange square is shown in the upper left-hand corner of the component, as shown in Figure 15–7.

Figure 15–7 contextInfo Displays a Square



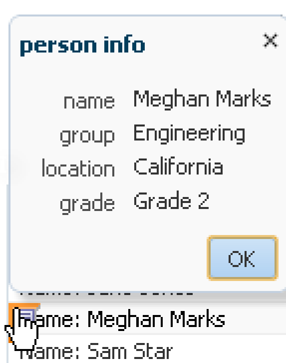
When the user places the cursor over the square, a larger triangle with a note icon and tooltip is displayed, indicating that additional information is available, as shown in Figure 15–8.

Figure 15–8 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 15–9.

Figure 15–9 Dialog launched From contextInfo Component



15.5.1 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 Section 15.2, "Declaratively Creating Popups."
3. You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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 Section 6.4, "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 Section 15.3.1, "How to Declaratively Invoke a Popup Using the af:showPopupBehavior Tag." For the **TriggerType**, make sure to enter `contextInfo`.

15.6 Controlling the Automatic Cancellation of Inline Popups

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

  For more information, see Section 15.2.1, "How to Create a Dialog."
- `panelWindow` component to create an inline window

  For more information, see Section 15.2.2, "How to Create a Panel Window."
- Menu components to create context menus

  For more information, see Section 15.2.3, "How to Create a Context Menu."
- `noteWindow` component to create a note window

  For more information, see Section 15.2.4, "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 more information about page rendering, see Chapter 8, "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 more information about the use of components that render toggle icons, see Section 9.9, "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.

15.6.1 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. For more information, see Section 15.6, "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. For more information, see Section 15.1.2, "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.

15.6.2 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 Example 15–12:

Example 15–12 Metadata to Prevent the Automatic Cancellation of an Inline Popup

```
<af:popup id="p1" autoCancel="disabled">
  ...
</af:popup>
```

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.

15.7 Resetting Input Fields in a Popup

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 more information about how the setting that you choose for the `contentDelivery` attribute determines the content delivery strategy for your `popup` component, see Section 15.2, "Declaratively Creating Popups" and Section 15.2.5, "What Happens at Runtime: Popup Component Events."

Example 15–13 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 Section 15.2.1, "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.

Example 15–13 The resetListener Tag on the Popup Component

```
<af:popup id="popup" contentDelivery="immediate">
  <af:resetListener type="popupCanceled"/>
</af:popup>
```

For more information about using the `resetListener` component independently of a `popup` component, see Section 20.5.2, "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 more information about setting the `resetEditableValues` attribute of the `popup` component, see Section 15.2.1, "How to Create a Dialog."

15.7.1 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. For more information, see Section 15.7, "Resetting Input Fields in a Popup."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 15.1.2, "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 Section 15.2, "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.

15.7.2 What Happens When You Configure a Popup to Reset Its Input Fields

JDeveloper writes entries similar to those shown in Example 15–14 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`.

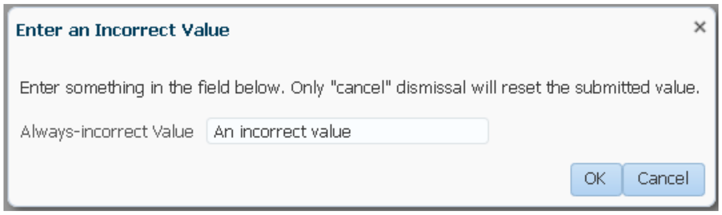
Example 15–14 Popup Component Configured to Reset Input Fields Using Reset Listener

```
<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>
```

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 15–10.

Figure 15–10 Popup Component Resetting Input Fields



16

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.

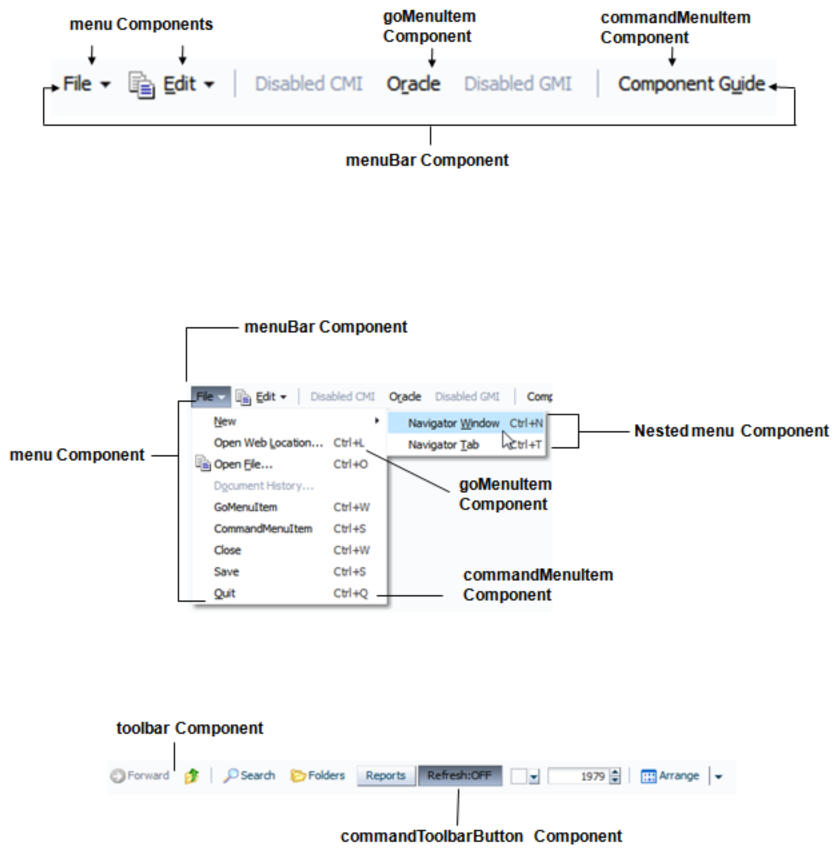
For information about creating navigation menus, that is, menus that allow you to navigate through a hierarchy of pages, see Section 20.6, "Using Navigation Items for a Page Hierarchy."

This chapter includes the following sections:

16.1 About Menus, Toolbars, and Toolboxes

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 16–1 shows the different components used to create menus and toolbars.

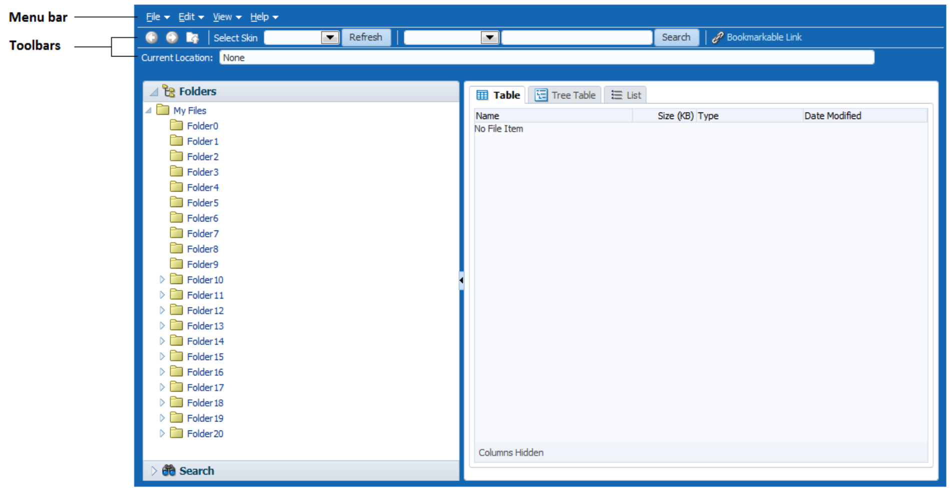
Figure 16–1 Menu and Toolbar Components



16.1.1 Menu Components Use Cases and Examples

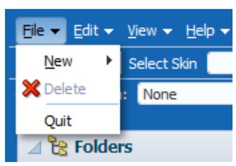
Menu components are used to create menus that allow users to add or edit items, search data, change the view, or launch help. For example, the ADF Faces demo application contains both a menu bar and a toolbar, as shown in Figure 16–2.

Figure 16–2 Menu Bar and Toolbar in File Explorer Application



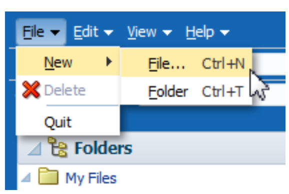
When a user chooses a menu item in the menu bar, the `menu` component displays a list of menu items, as shown in Figure 16–3.

Figure 16–3 Menu in the File Explorer Application



As shown in Figure 16–4, menus can be nested.

Figure 16–4 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 Section 12.9, "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 Section 15.2.3, "How to Create a Context Menu."

16.1.2 Additional Functionality for Menu and Toolbar Components

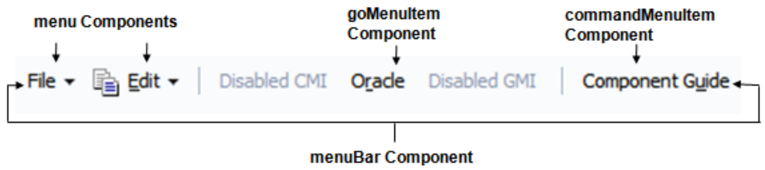
You may find it helpful to understand other ADF Faces features before you implement your menu and toolbar components. Additionally, once you have added these components to the page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality 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. For more information, see Section 20.5, "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). For more information, see Section 12.9, "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. For more information, see Section 15.2.3, "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. For more information, see Section 3.5.2, "How to Use the EL Format Tags."
- **Events**: You can use command menu components to launch action events. For more information about events, see Chapter 6, "Handling Events." For more information about action events specifically, see Section 20.3, "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 Section 33.3.4, "How to Define Access Keys for an ADF Faces Component."For more information about accessibility, see Section 33.3, "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. For more information, see Chapter 32, "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. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

16.2 Using Menus in a Menu Bar

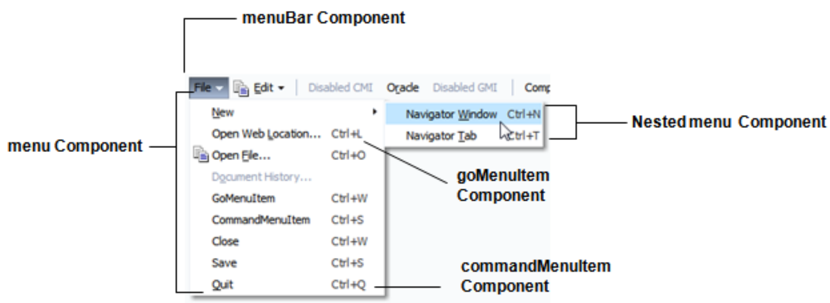
Use the `menuBar` component to render a bar that contains the menu bar items (such as **File** in the File Explorer application). These items can be `menu` components, which hold a vertical menu, as well as `commandMenuItem` components that invoke some operation on the application, and `goMenuItem` components that invoke a URL, as shown in Figure 16–5.

Figure 16–5 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 16–6.

Figure 16–6 Components Used to Create a Menu



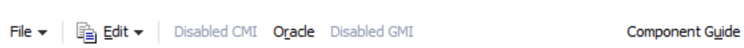
You can use more than one menu bar by enclosing them in a toolbox. Enclosing them in a toolbox stacks the menu bars so that the first menu bar in the toolbox is displayed at the top, and the last menu bar is displayed at the bottom. When you use more than one menu bar in a single toolbox row (by having 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 16–7, 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 16–7 Spacer Component Stretches and Shrinks

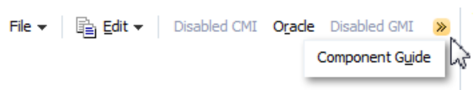


When a window is resized such that all the components within the menu bar can no longer be displayed, the menu bar displays an overflow icon, identified by the arrow cursor as shown in Figure 16–8.

Figure 16–8 Overflow Icon in a Menu Bar

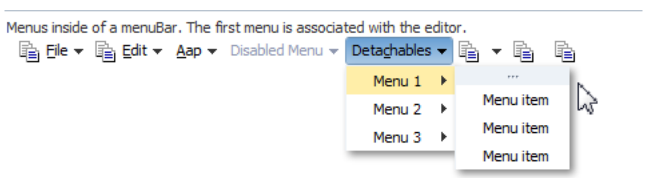
Clicking that overflow icon displays the remaining components in a popup window, as shown in Figure 16–9.

Figure 16–9 menu Component in an Overflow Popup Window



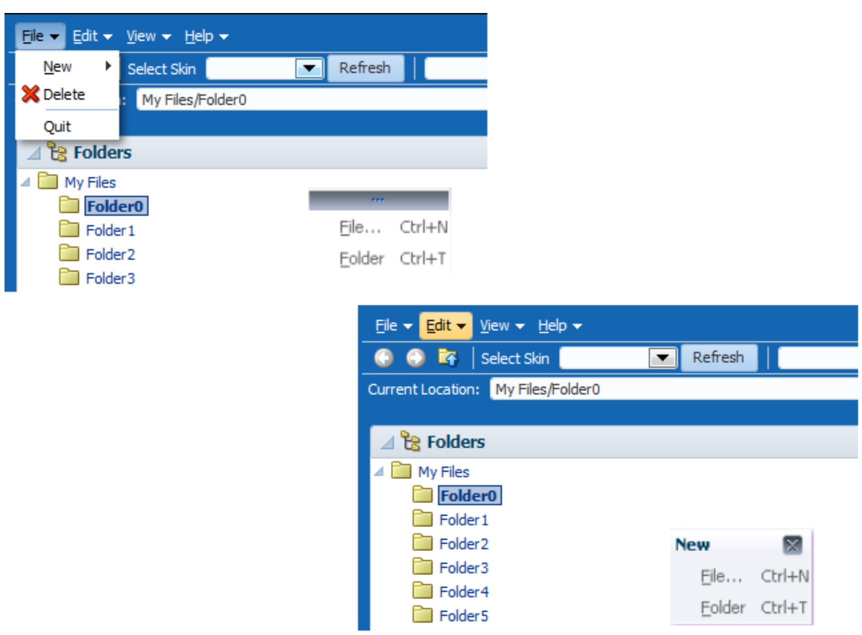
Menus and submenus can be made to be detachable and to float on the browser window. Figure 16–10 shows the **Menu 1** submenu in the **Detachables** menu configured to be detachable. The top of the menu is rendered with a bar to indicate that it can be detached.

Figure 16–10 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 16–11.

Figure 16–11 Floating Detached Menu

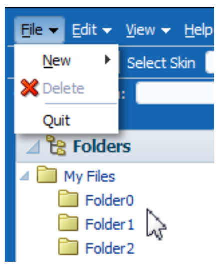


Tip: Consider using detachable menus when you expect users to:

- 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.

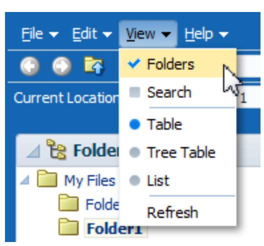
The `menu` and `commandMenuItem` components can each include an icon image. Figure 16–12 shows the **Delete** menu item configured to display a delete icon (a red *x*).

Figure 16–12 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 16–13 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 16–13 Icons and Radio Buttons Denote the Chosen Menu Items



You can also configure a `commandMenuItem` component to have an antonym. Antonyms display different text when the user chooses a menu item. Figure 16–14 shows an **Open** menu item in the **Special** menu.

Figure 16–14 The Edit Menu of the File Explorer Application

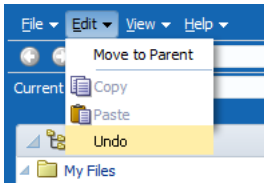
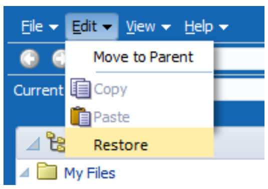


Figure 16–14 shows the `commandMenuItem` component for the **Undo** menu item configured to be an antonym. When the user chooses **Undo**, the next time the user returns to the menu, the menu item will display the antonym **Restore**, as shown in Figure 16–15.

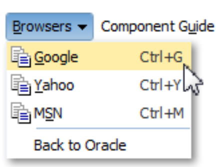
Figure 16–15 Menu Items Can Be Antonyms



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 16–16 shows three `goMenuItem` components used to navigate to external web sites.

Figure 16–16 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. For more information, see Section 15.2.3, "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. For more information, 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 Section 20.7, "Using a Menu Model to Create a Page Hierarchy."

16.2.1 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 Section 12.9, "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. For more information, see Section 16.2, "Using Menus in a Menu Bar."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 16.1.2, "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 more information about toolboxes, see Section 16.3, "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 Example 16–4, `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`.

Example 16–1 Flex Attribute Determines Length of Toolbars

```
<af:toolbox>
  <af:menuBar id="menuBar1" flex="0">
    <af:menu text="MenuA"/>
  </af:menBar>
  <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: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. For more information, see Section 16.3.2, "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 Section 16.3.2, "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 more information about access keys and the ampersand notation, see Section 33.3, "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 16–10), 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 16–11).

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.

   Example 16–2 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.

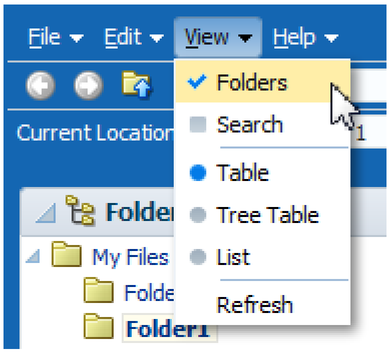
Example 16–2 Grouping Menu Items

```
<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 16–17 shows how the menu will be displayed when it is first accessed.

Note: By default, ADF Faces components use the Skyros skin. You can change this by creating your own skin. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

Figure 16–17 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. For more information, see Chapter 31, "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 16–4).

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.
- **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 16–4.

> **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, &Save sets the menu item label to **Save**, and at the same time sets the menu item access key to the letter S. For more information about access keys and the ampersand notation, see Section 33.3, "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 more information about configuring navigation in your application, see Section 3.3, "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 Section 33.3.4, "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 Example 16–3.

Example 16–3 Using a Spacer to Justify menuBar Components

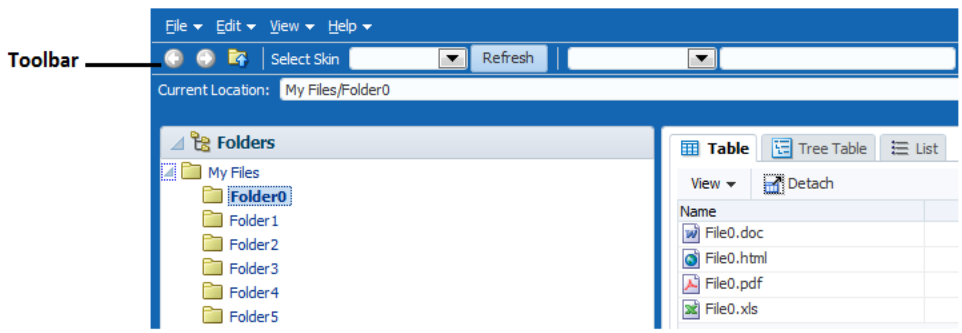
```
<af:menuBar binding="#{editor.component}" id="menuBar1" stretchId="stretch1">
  <af:menu text="File" id="m1">
. . .
  </af:menu>
. . .
  <af:commandMenuItem text="Disabled CMI"/>
  <af:goMenuItem textAndAccessKey="O&amp;racle destination="http://www.oracle.com"
                id="gmi1"/>
  <af:goMenuItem text="Disabled GMI" destination="http://www.gizmo.com"
                shortDesc="disabled goMenuItem" id="gmi2"/>
  <af:spacer id="stretch1" clientComponent="true"/>
  <af:commandMenuItem textAndAccessKey="Component G&amp;uide"
                     action="guide" id="cmi9"/>
</af:menuBar>
```

16.3 Using Toolbars

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 16–18 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 16–18 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 16–18. 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.

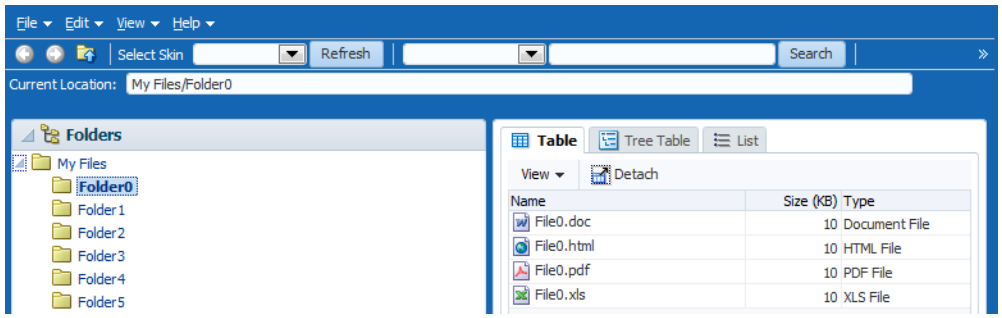
If you wish toolbars to be displayed next to each other (rather than stacked), you can enclose them in a `group` component.

> **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.

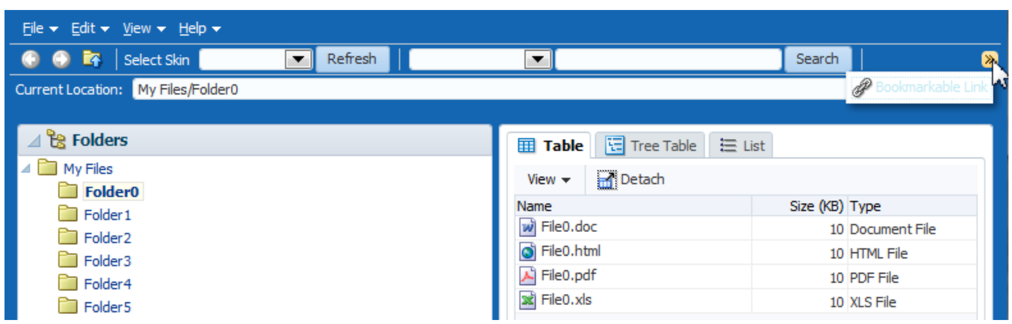
However, because no component in the top toolbar is set to stretch, it does not change size when the window is resized. When a window is resized such that all the components within the toolbar can no longer be displayed, the toolbar displays an overflow icon, identified by an arrow cursor in the upper right-hand corner, as shown in Figure 16–19.

Figure 16–19 Overflow Icon in a Toolbar



Clicking that overflow icon displays the remaining components in a popup window, as shown in Figure 16–20.

Figure 16–20 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.

16.3.1 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. For more information, see Section 16.3, "Using Toolbars."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 16.1.2, "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 Example 16–4, `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`.

Example 16–4 Flex Attribute Determines Length of Toolbars

```
<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. For more information, see Section 16.3.2, "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 Section 16.3.2, "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 16–17.
>
> 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 more information about configuring navigation, see Section 3.3, "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 Section 16.2.1, "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 Example 16–5.

Example 16–5 Using the stretchId Attribute

```
<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 Example 16–6.

Example 16–6 Using a Spacer to Justify Toolbar Components

```
<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>
```

16.3.2 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. For more information, see Section A.6.2.6, "Language Reading Direction."

16.3.3 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.

17
Using a Calendar Component

This chapter describes how to use the ADF Faces calendar component to create a calendar application.

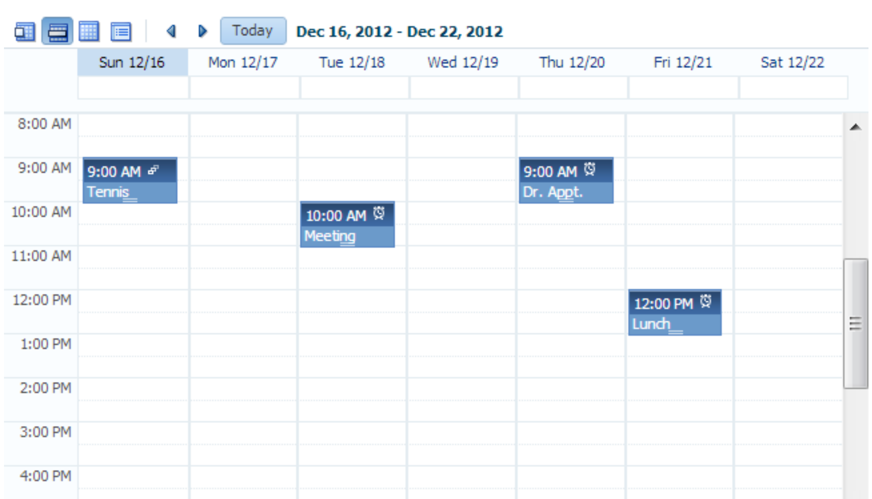
This chapter includes the following sections:

- Section 17.1, "About Creating a Calendar Component"
- Section 17.2, "Creating the Calendar"
- Section 17.3, "Configuring the Calendar Component"
- Section 17.4, "Adding Functionality Using Popup Components"
- Section 17.5, "Customizing the Toolbar"
- Section 17.6, "Styling the Calendar"

17.1 About Creating a Calendar Component

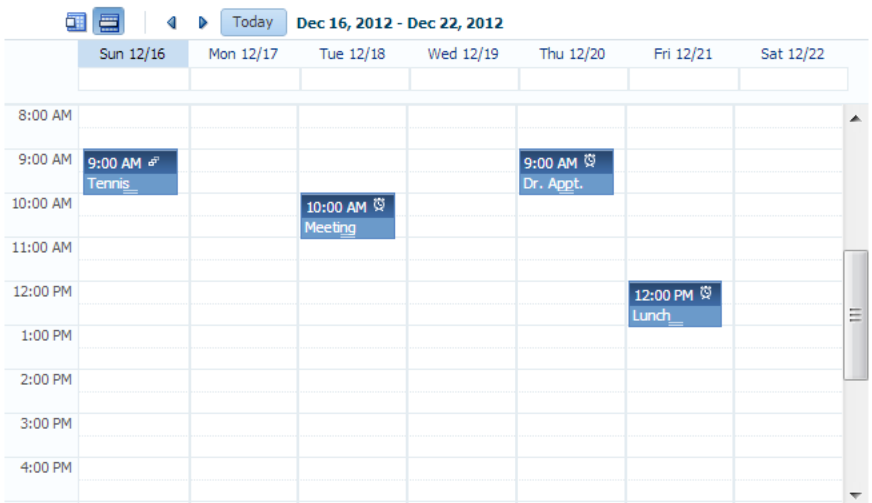
ADF Faces includes a calendar component that by default displays activities in daily, weekly, monthly, or list views for a given provider or providers (a provider is the owner of an activity). Figure 17–1 shows an ADF Faces calendar in weekly view mode with some sample activities.

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. For more information, see Section A.6, "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. For more information, see Section 17.3, "Configuring the Calendar Component."

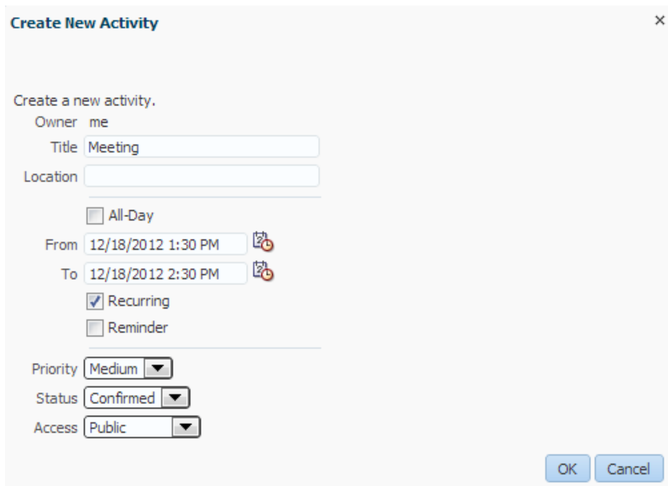
The calendar includes a toolbar with built-in functionality that enables a user to change the view (between daily, weekly, monthly, or list), go to the previous or next day, week, or month, and return to today. The toolbar is fully customizable. You can choose which buttons and text to display, and you can also add buttons or other components. For more information, see Section 17.5, "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. For more information, see Chapter 35, "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. For more information, see the "Allowing User Customizations at Runtime" chapter of *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



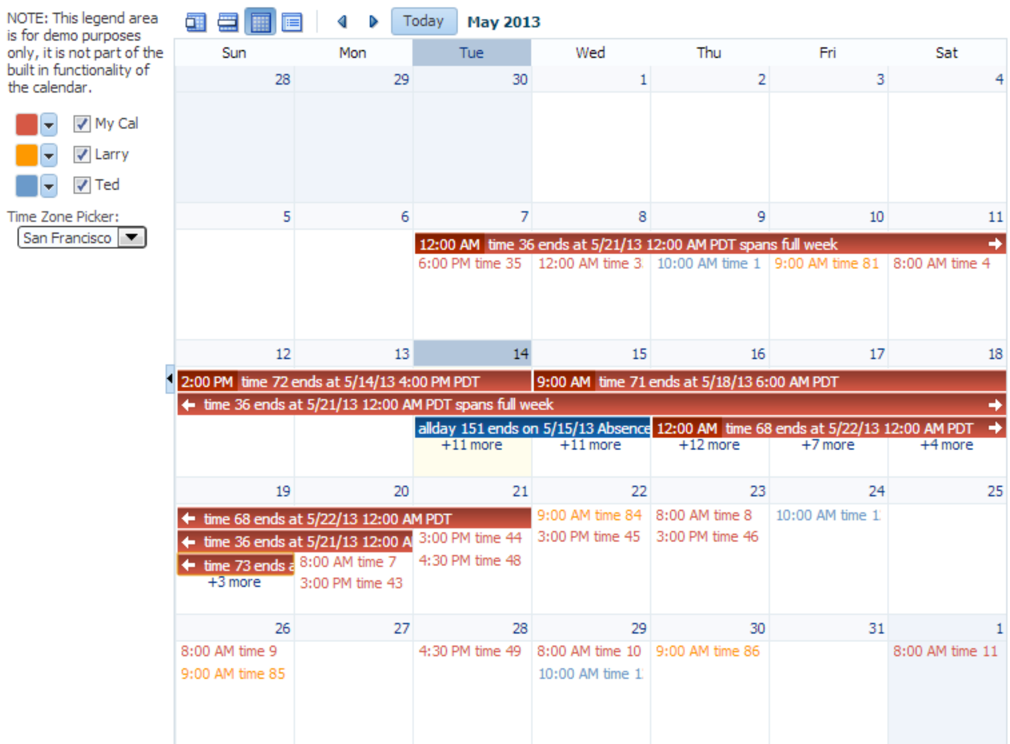
For more information about implementing additional functionality using events, facets, and popup components, see Section 17.4, "Adding Functionality Using Popup Components."

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 more information about adding drag and drop functionality, see Section 36.7, "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



17.1.1 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, then 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. For more information, see the "Using the ADF Faces Calendar Component" section 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. For more information, see Section 17.2, "Creating the Calendar."

17.1.2 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 Chapter 4, "Using ADF Faces Client-Side Architecture."
- **JavaScript APIs**: All components have JavaScript client APIs that you can use to set or get property values. For more information, 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. For more information, see Chapter 6, "Handling Events."
- You can display tips and messages, as well as associate online help with a calendar component. For more information, see Chapter 19, "Displaying Tips, Messages, and Help."
- You may want other components on a page to update based on selections you make from a calendar component. For more information, see Section 5.3, "Using the Optimized Lifecycle."
- You can change the appearance using skins. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."
- You can make your components accessible. For more information, see Chapter 33, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."
- You can create popups for additional functionality. For more information about using these events to provide additional functionality, see Section 17.4, "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). For more information, see the "Creating a Basic Databound Page" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

17.2 Creating the Calendar

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. For more information, see the "Using the ADF Faces Calendar Component" section 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.

17.2.1 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.

17.2.2 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. For more information, see Section 17.2.1, "Calendar Classes."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "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 more 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 demo application (for more information about downloading and installing the demo application, see Chapter 2, "ADF Faces Demo Application").

For more information about creating managed beans, see Section 3.6, "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 more 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 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). For more information, see Section 17.6.1, "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 more 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 more information about the default height and width, see Section 17.3, "Configuring the Calendar Component." For more information about stretching components, see Section 9.2.1, "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.

17.3 Configuring the Calendar Component

Configure the many display attributes for the calendar, for example, the time displayed at the beginning of a day.

17.3.1 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. For more information, see Section 17.3, "Configuring the Calendar Component."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "Additional Functionality for the Calendar."

To configure a calendar:

1. 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.

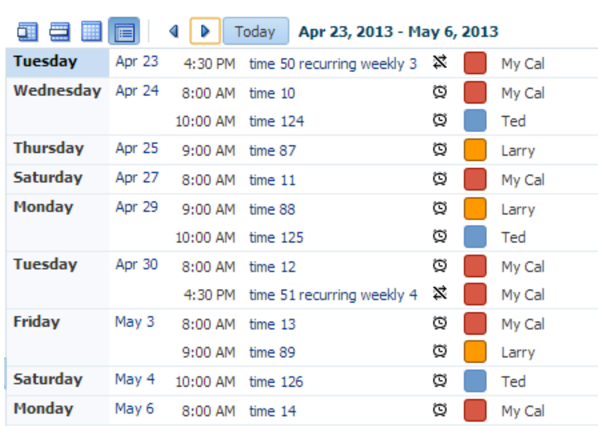
     Note: In order to handle an overflow of tasks for a given day in the `month` view, if you enter `month` and do not also enter `all`, then you must also enter `day`.

   - **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



2. 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. For more information, see Section A.6, "Configuration in trinidad-config.xml."

3. 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:

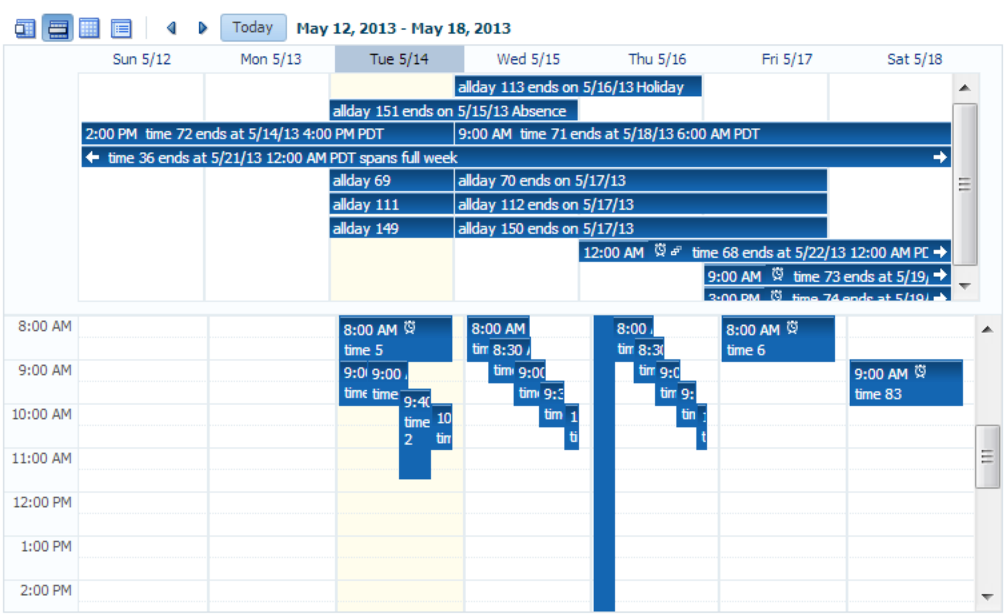
     – Activities with the `holiday` tag

     – 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.

4. 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. For more information, see Section 36.7, "Adding Drag and Drop Functionality to a Calendar."

You can now add the following functionality:

- Create, edit, and delete activities using popup components. For more information, see Section 17.4, "Adding Functionality Using Popup Components."
- Move activities around on the calendar. For more information, see Section 36.7, "Adding Drag and Drop Functionality to a Calendar."
- Change or add to the toolbar buttons in the toolbar. For more information, see Section 17.5, "Customizing the Toolbar."
- Change the appearance of the calendar and events. For more information, see Section 17.6, "Styling the Calendar."

17.3.2 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 more information about using these events to provide additional functionality, see Section 17.4, "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 rerendered automatically because the calendar component becomes a partial page rendering (PPR) target, triggering an immediate refresh.

17.4 Adding Functionality Using Popup Components

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.	`CalendarActivityEvent`	`activityContextMenu`: 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).	`menu`
Select an activity and press the Delete key.	`CalendarActivityEvent`	`activityDelete`: The enclosed popup component can be used to display a dialog that allows the user to delete the selected activity.	`dialog`
Click or double-click an activity, or select an activity and press the Enter key.	`CalendarActivityEvent`	`activityDetail`: The enclosed popup component can be used to display the activity's details.	`dialog`
Hover over an activity.	`CalendarActivityEvent`	`activityHover`: The enclosed popup component can be used to display high-level information about the activity.	`noteWindow`
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`

Table 17–1 (Cont.) Calendar Faces Events and Associated Facets

User Action	Event	Associated Facet	Component to Use in Popup
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`

17.4.1 How to Add Functionality Using Popup Components

To add functionality, create the popups and associated components in the associated facets.

Before you begin:

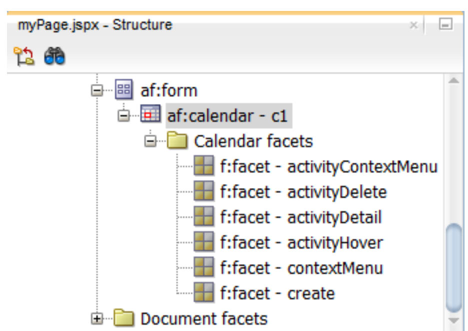
It may be helpful to have an understanding of popup components. For more information, see Section 17.4, "Adding Functionality Using Popup Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "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 more information about creating popup components, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

   Example 17–1 shows the JSF code for a dialog popup component used in the `activityDelete` facet.

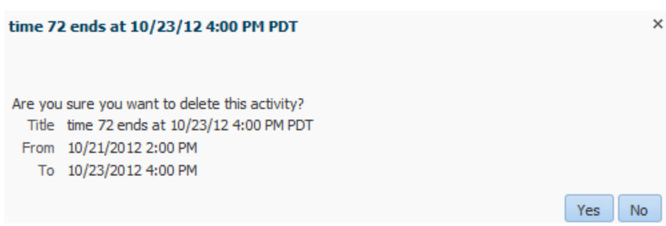
Example 17–1 JSF Code for an Activity Delete Dialog

```
<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. Example 17–2 shows the `calendarActivityListener` for the `calendar.jspx` page in the ADF Faces demo application.

Example 17–2 calendarActivityListener Handler

```
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 more information about creating dialogs and other popup components, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

17.5 Customizing the Toolbar

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

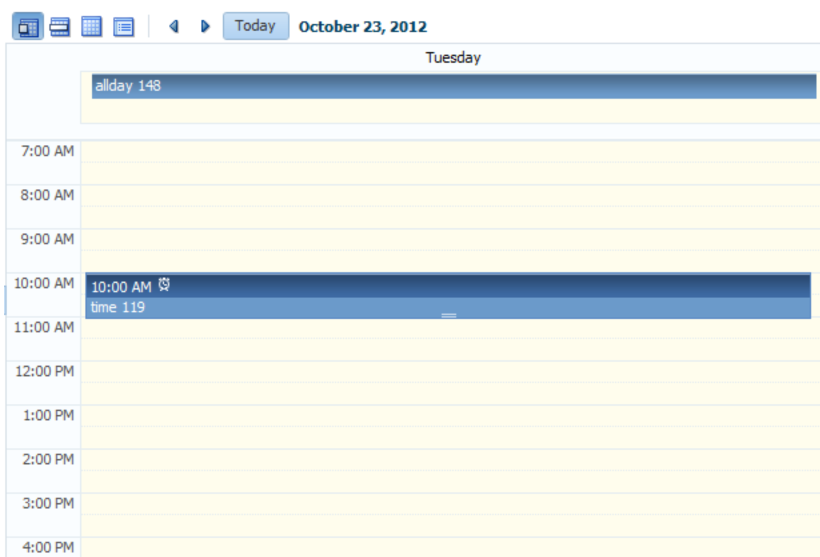
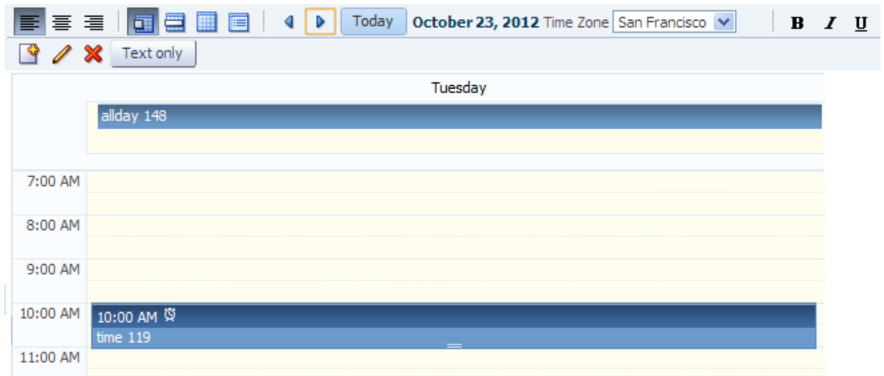


Figure 17–10 shows a toolbar that has been customized. It has added toolbar buttons, including buttons that are right-aligned on the top toolbar, and buttons in a second toolbar.

Figure 17–10 Customized Toolbar for a Calendar



17.5.1 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. For more information, see Section 17.5, "Customizing the Toolbar."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "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 more information about toolbars and toolbar buttons, see Section 16.3, "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 `toolboxLayout` 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 `toolboxLayout` 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. Example 17–3 shows the value of the `toolboxLayout` 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.

Example 17–3 Value for Custom Toolbar

```
<af:calendar binding="#{editor.component}" id="calendar1"
             value="#{calendarBean.calendarModel}"
             timeZone="#{calendarBean.timeZone}"
             toolboxLayout="customToolbarAlign all customToolbarTZ stretch
                           customToolbarBold newline customToolbarCreate"
. . .
```

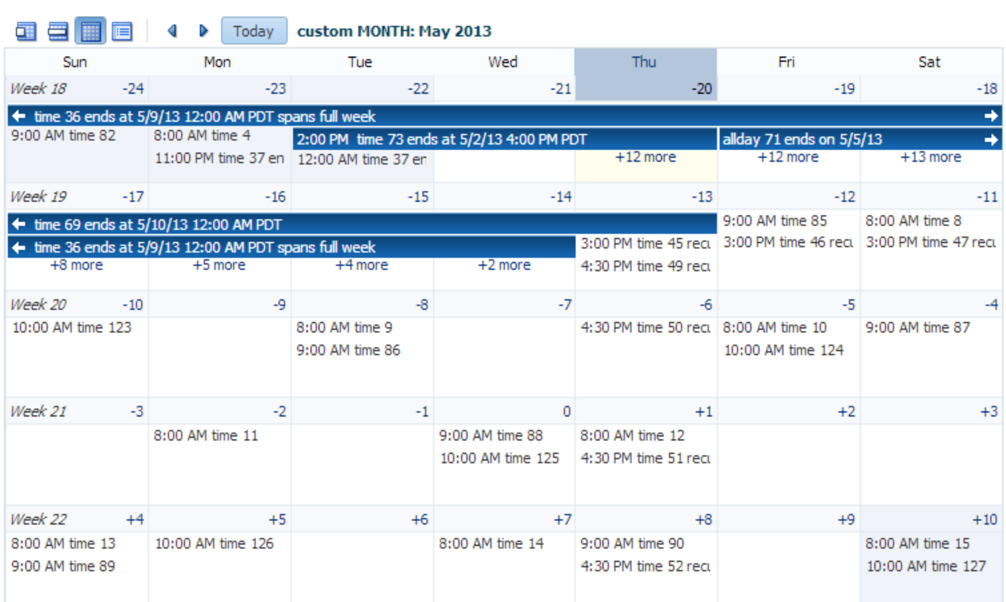
```
        <f:facet name="customToolbarAlign">
          <af:toolbar>
            <af:button id="alignLeft" shortDesc="align left"
                                         icon="/images/alignleft16.png" type="radio"
                                         selected="true"/>
            <af:button id="alignCenter" 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>
```

17.6 Styling the Calendar

Like other ADF Faces components, the calendar component can be styled as described in Chapter 31, "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.

Figure 17–11 Customized Display of Dates in a Calendar

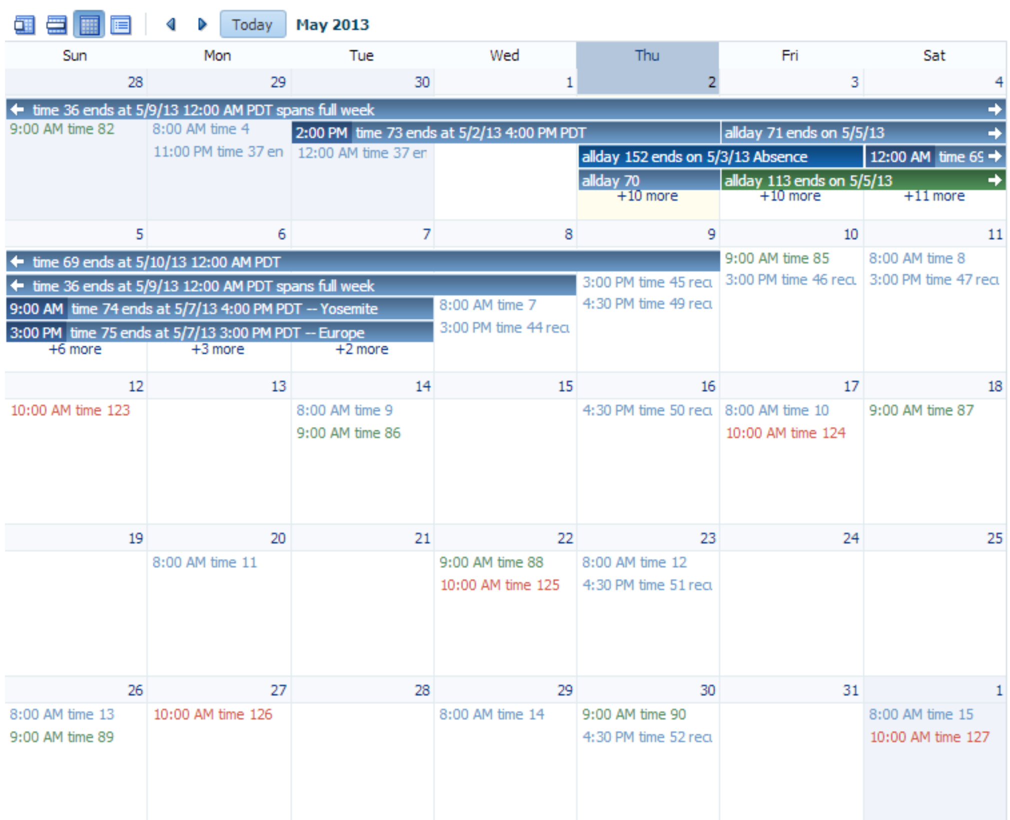


17.6.1 How to Style Activities

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. For more information, see Section 17.6, "Styling the Calendar."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "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 more information about implementing the `CalendarActivity` class, see Section 17.2.2, "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)
`{"Me"}`	`CalendarActivityRamp.getActivityRamp (CalendarActivityRamp.RampKey.RED)`
`{"LE"}`	`CalendarActivityRamp.getActivityRamp (CalendarActivityRamp.RampKey.ORANGE)`
`{"TF"}`	`CalendarActivityRamp.getActivityRamp (CalendarActivityRamp.RampKey.BLUE)`

3. In the Structure window, select the calendar component, and in the Properties window, bind the `activityStyles` attribute to the map.

17.6.2 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.

17.6.3 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.

Before you begin:

It may be helpful to have an understanding of calendar styling. For more information, see Section 17.6, "Styling the Calendar."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 17.1.2, "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::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::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::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.

   Example 17–4 shows the DemoDateCustomizer class that displays the week number in the first day of the week, and instead of the day of the month, a countdown number to a specific date, as shown in Figure 17–11.

Example 17–4 Date Customizer Class

```
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;

  }

}
```

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.

18

Using Output Components

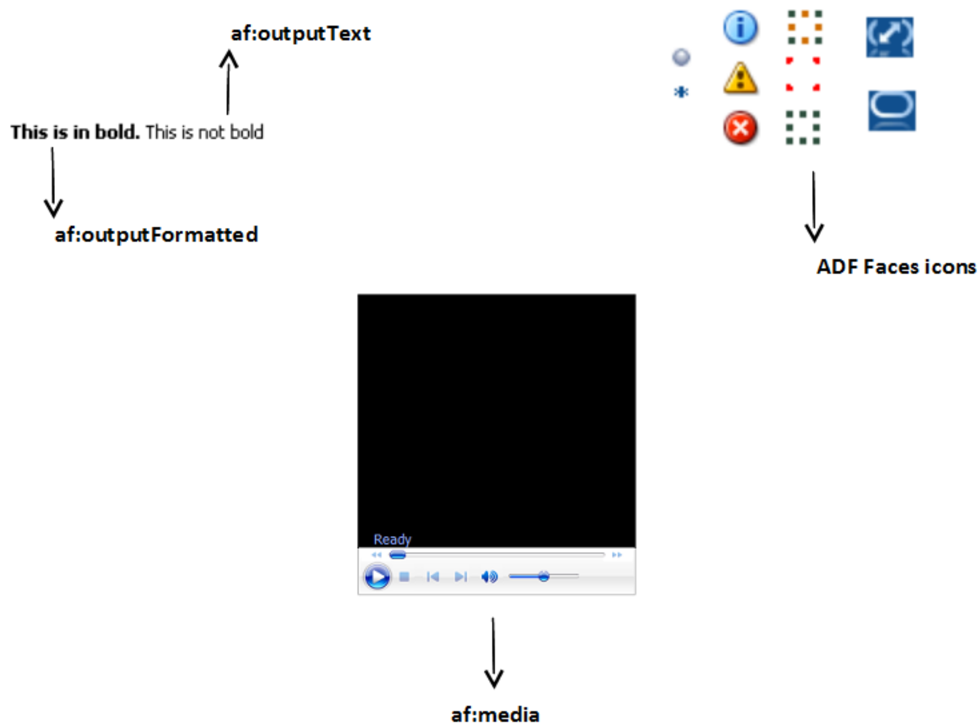
This chapter describes how to use the ADF Faces `outputText`, `outputFormatted`, `image`, `icon`, and `statusIndicator` components to display output text, images, and icons, and how to use the `media` component to enable users to play video and audio clips.

This chapter includes the following sections:

18.1 About Output Text, Image, Icon, and Media Components

ADF Faces provides components for displaying text, icons, and images, and for playing audio and video clips on JSF pages.

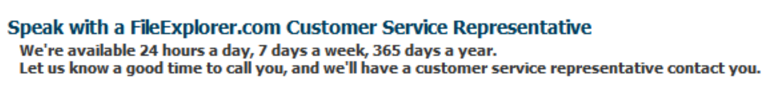
Figure 18–1 ADF Faces Output Components



18.1.1 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



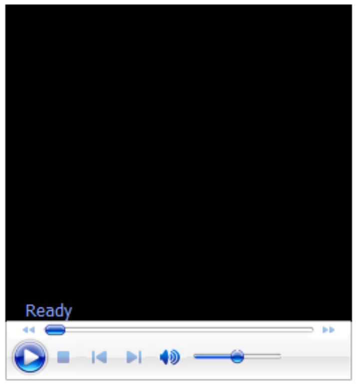
Many ADF Faces components can have icons associated with them. For example, in a menu, each of the menu items can have an associated icon. You identify the image to use for each one as the value of an `icon` attribute for the menu item component itself. Information and instructions for adding icons to 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



18.1.2 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. For more information, see Section 3.5.2, "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 more information about conversion, see Chapter 7, "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. For more information, see Chapter 36, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."
- **Skins**: You can change the look and feel of output components by changing the skin. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

18.2 Displaying Output Text and Formatted Output Text

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` property. For example:

```
<af:outputText value="The submitted value was: "/>
```

Example 18–1 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 more information about conversion, see Section 7.3, "Adding Conversion").

Example 18–1 Output Text

```
<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.

Example 18–2 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`.

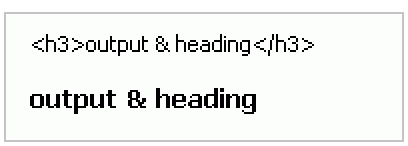
Example 18–2 Output Text With and Without the escape Property Set

```
<af:outputText value="&lt;h3>output &amp; heading&lt;/h3>"/>
<af:outputText value="&lt;h3>output &amp; heading&lt;/h3>"
               escape="false"/>
```

CAUTION: You should 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



As with the `outputText` component, the `outputFormatted` component also displays the text specified for the `value` property, 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 more information about using inline styles and creating skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

Example 18–3 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).

Example 18–3 Using outputFormatted to Bold Some Text for a JSPX file

```
<af:outputFormatted value="&lt;b>This is in bold.&lt;/b> This is not bold"/>
```

Example 18–4 Using outputFormatted to Bold Some Text for a JSP file

```
<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

18.2.1 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. For more information, see Section 18.2, "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. For more information, see Section 18.1.2, "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.

2. 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

> **Note:** If the `styleUsage` and `styleClass` attributes are both set, the `styleClass` attribute takes precedence.

18.2.2 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
 	Line break
<hr>	Horizontal rule
... ...	Lists: ordered list, unordered list, and list item
...	Paragraph
...	Bold
<i>...</i>	Italic
<tt>...</tt>	Teletype or monospaced
<big>...</big>	Larger font
...	Smaller font
<pre>...</pre>	Preformatted: layout defined by whitespace and line break characters preserved
...	Span the enclosed text
<a>...	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
<	Less than

Table 18–2 (Cont.) Character Codes for Use in af:outputFormatted Values

Character Code	Character
>	Greater than
&	Ampersand
®	Registered
©	Copyright
	Nonbreaking space
"	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.

18.3 Displaying Icons

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. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

18.3.1 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. For more information, see Section 18.3, "Displaying Icons."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 18.1.2, "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.

18.4 Displaying Images

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. For more information, see Section 18.5, "Using Images as Links."

18.4.1 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. For more information, see Section 18.1.2, "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.

18.5 Using Images as Links

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 more information about the `link` component, see Section 20.3, "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.

18.5.1 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. For more information, see Section 18.5, "Using Images as Links."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 18.1.2, "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**.

18.6 Displaying Application Status Using Icons

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. For more information, see the "Using the Active Data Service" chapter of *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.
	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 more information about using skins, see Chapter 31, "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. For more information, see Section 18.6, "Displaying Application Status Using Icons."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 18.1.2, "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.

18.7 Playing Video and Audio Clips

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.

18.7.1 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. For more information, see Section 18.7, "Playing Video and Audio Clips."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 18.1.2, "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.

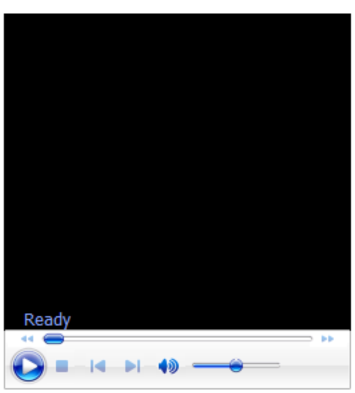
As an example, Example 18–5 uses the `all` setting for a `media` component.

Example 18–5 Controls for a Media Player

```
<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 with 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.

Example 18–6 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.

Example 18–6 Media Component to Play a Video Clip Continuously

```
<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

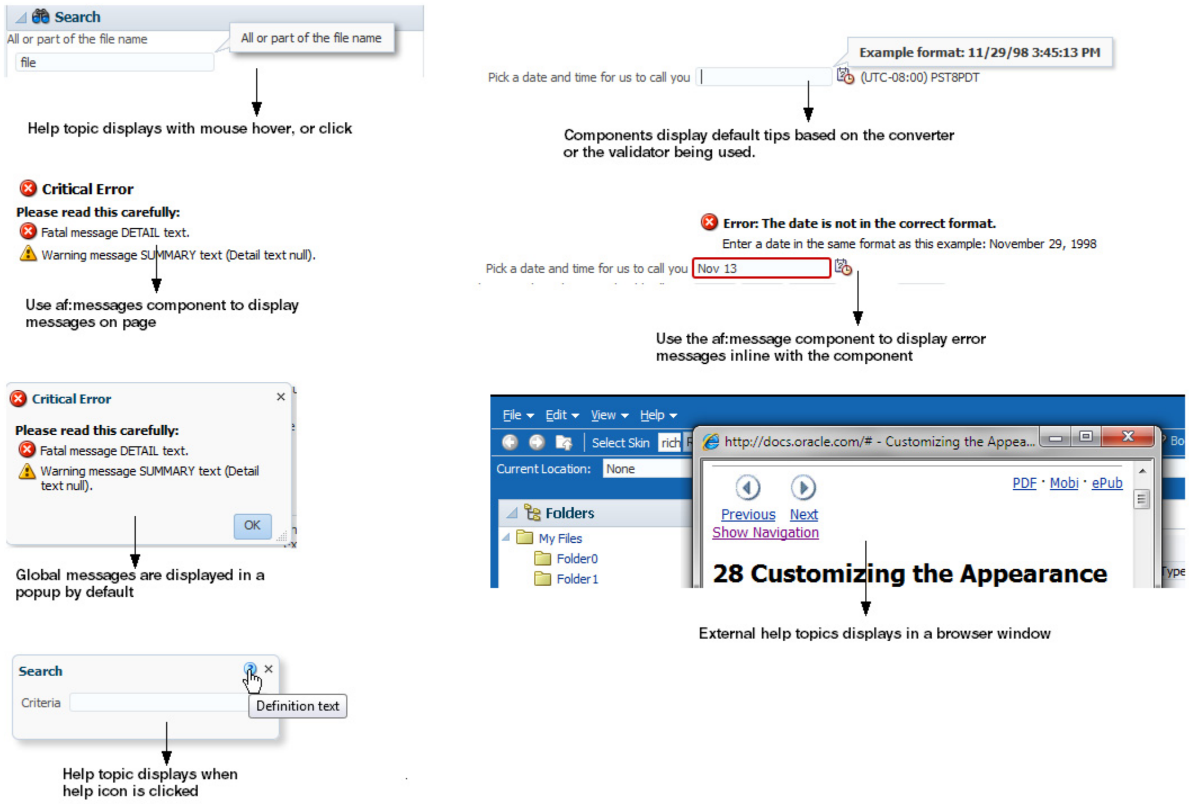
This chapter describes how to use the `shortDesc` attribute to display tips, and how to display messages for ADF Faces components. This chapter also describes how to provide different levels of help information for users.

This chapter includes the following sections:

19.1 About Displaying Tips and Messages

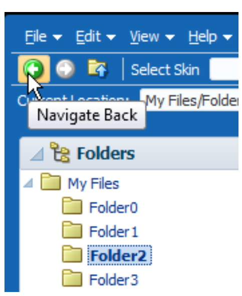
ADF Faces provides many different ways for displaying informational text in an application. You can create simple tip text, validation and conversion tip text, validation and conversion failure messages, as well as elaborate help systems.

Figure 19–1 ADF Messaging Components



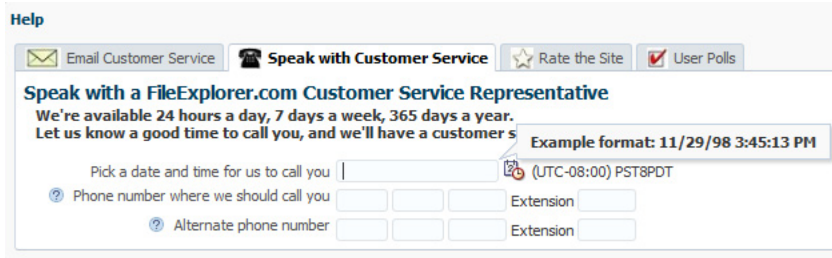
Many ADF Faces components support the `shortDesc` attribute, which for most components, displays tip information when a user hovers the cursor over the component. Figure 19–2 shows a tip configured for a toolbar button.

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.

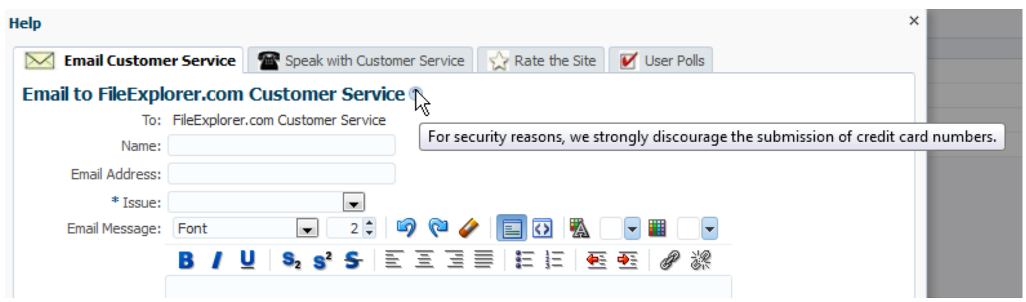
Figure 19–3 Hint Message Displays Format of Input Date



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:

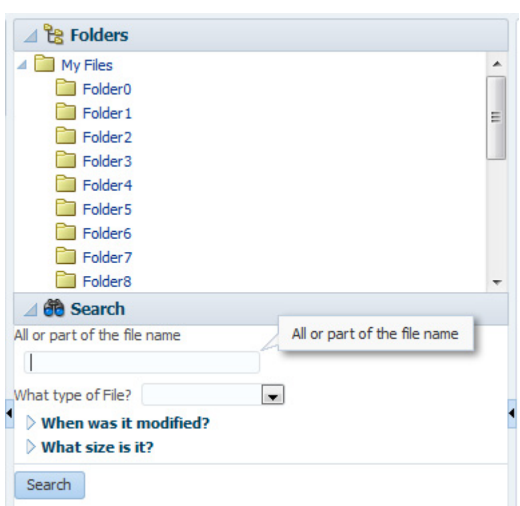
- Definition: Provides a help icon (question mark in a blue circle) with the help text appearing when the user mouses over the icon, as shown in Figure 19–4.

Figure 19–4 Definition Messages Display When Mousing Over the Icon



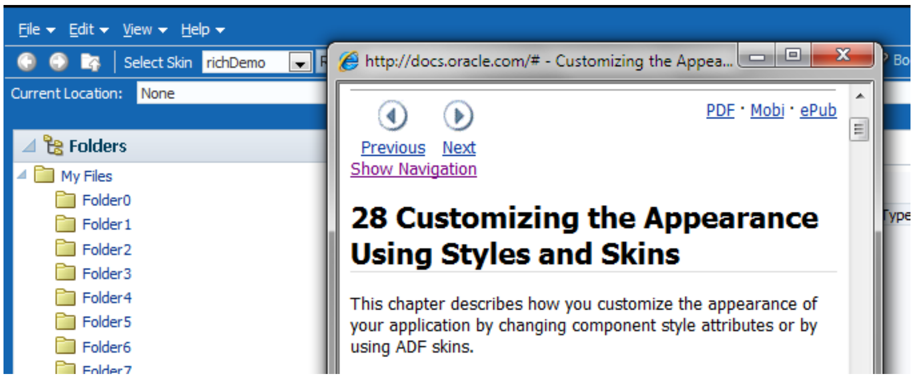
- Instruction: Depending on the component, this type of help either provides instruction text within the component (as with `panelHeader` components), or displays text in the note window that is opened when the user clicks in the component, as shown in Figure 19–5. The text can be of any length.

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 more information about creating help systems, see Section 19.5, "Displaying Help for Components."

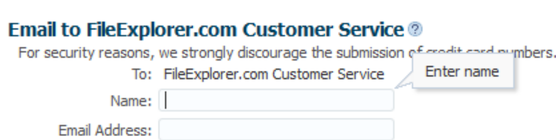
19.1.1 Messaging Components Use Cases and Examples

Messages can typically be divided into to 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



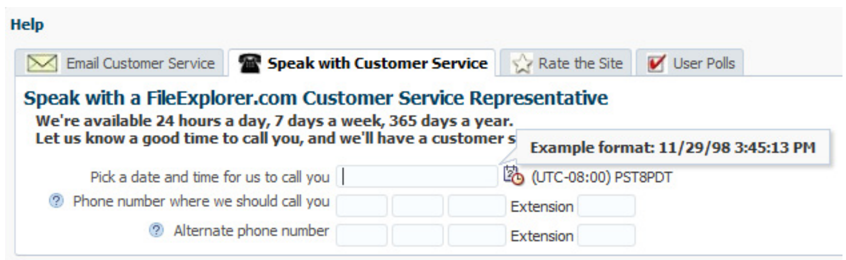
For more information about tooltips, see Section 19.2, "Displaying Tips for Components."

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.

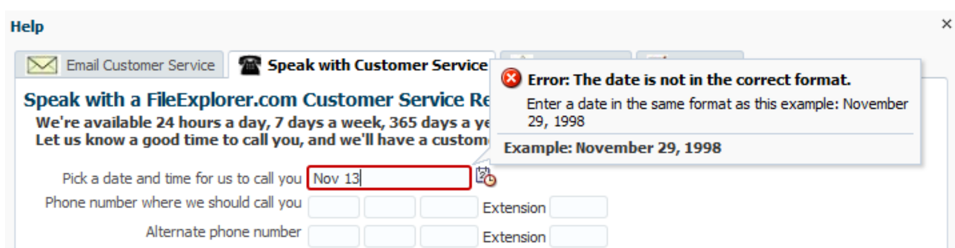
For example, when users click **Help > Give Feedback** in the File Explorer application, a dialog displays where they can enter a time and date for a customer service representative to call. Because the `inputDate` component contains a converter, when the user clicks in the field, a note window displays 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



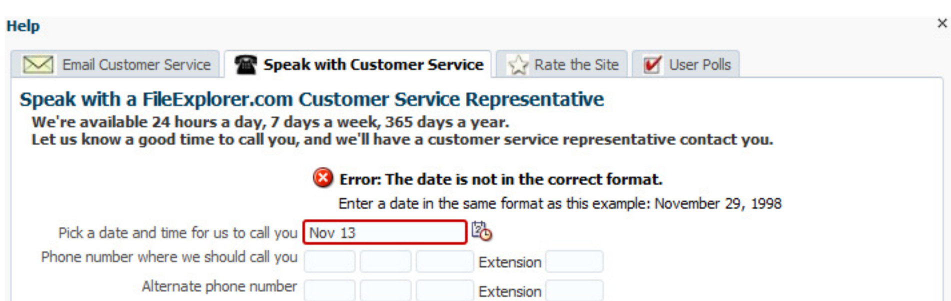
If a user enters a date incorrectly in the field shown in Figure 19–8, an error message is displayed, as shown in Figure 19–9. Note that the error message appears in the note window along with the hint.

Figure 19–9 Validation and Conversion Errors Display in Note Window



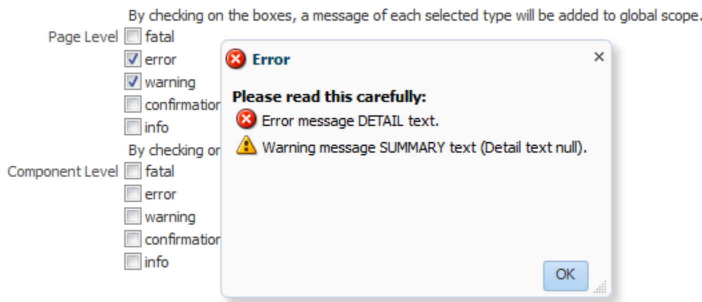
If you want to display an error message for a non-ADF Faces component, or if you want the message to be displayed inline instead of the note window, use the ADF Faces `message` component. When you use this component, messages are displayed next to the component, as shown in Figure 19–10.

Figure 19–10 Use the message Component to Display Messages Inline



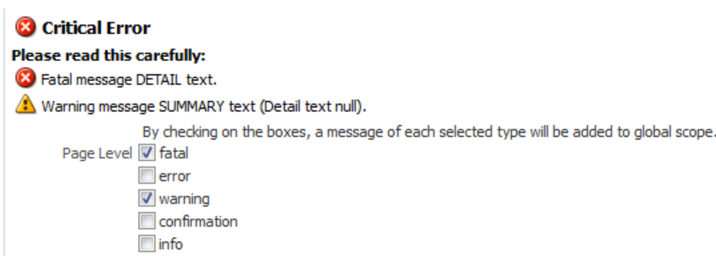
Global messages are by default displayed in a dialog, as shown in Figure 19–11. You do not need to add the popup component to the page.

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.

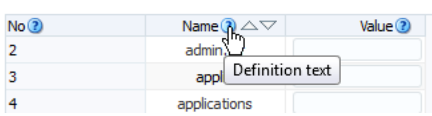
Figure 19–12 Use the messages Component to Display Global Messages on the Page



For more information about error messages, see Section 19.3, "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.

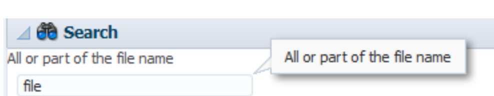
Figure 19–13 Definition Help for a Column Component



For more information about definition help, see Section 19.5, "Displaying Help for Components."

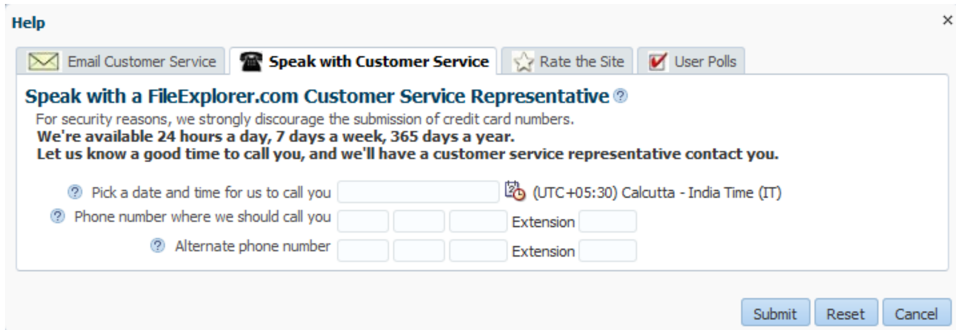
When you want to display field-level help, configure an input component to use instruction text. When the user clicks in the component, the help text is displayed in a note window, as shown in Figure 19–14.

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 more information about instruction and external help, see Section 19.5, "Displaying Help for Components."

19.1.2 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. For more information, see Section 3.5.2, "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 more information about client-side events, see Section 6.3, "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. For more information, see Chapter 31, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."

19.2 Displaying Tips for Components

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 Section 19.5, "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

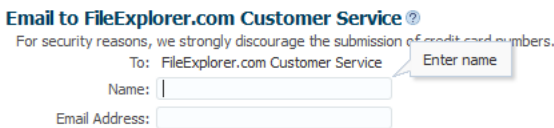
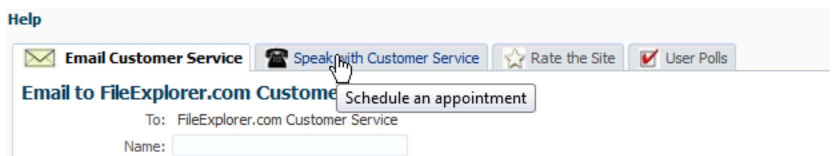


Figure 19–17 shows a tip as displayed for a `showDetailItem` component.

Figure 19–17 Tip for a showDetailItem Component



19.2.1 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. For more information, see Section 19.2, "Displaying Tips for Components."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 19.1.2, "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.

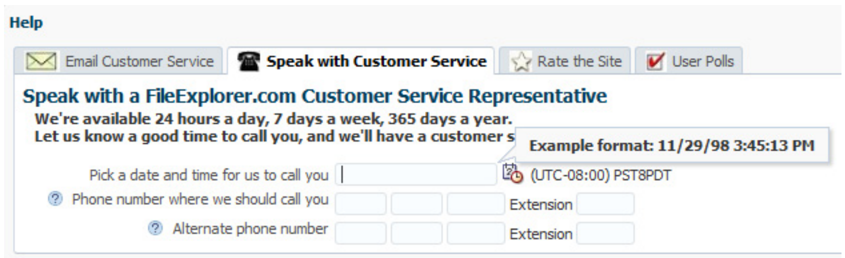
   If the text to be used is stored in a resource bundle, use the dropdown list to select **Select Text Resource**. Use the Select Text Resource dialog to either search for appropriate text in an existing bundle, or to create a new entry in an existing bundle. For more information about using resource bundles, see Chapter 32, "Internationalizing and Localizing Pages."

19.3 Displaying Hints and Error Messages for Validation and Conversion

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.

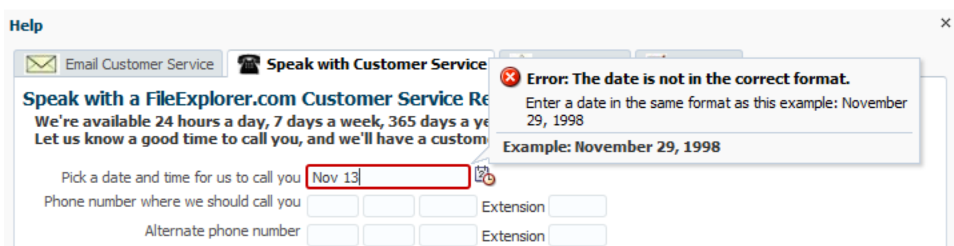
Figure 19–18 Validators and Converters Have Built-in Messages



When the value of an ADF Faces component fails validation, or cannot be converted by a converter, the component displays the resulting `FacesMessage` instance.

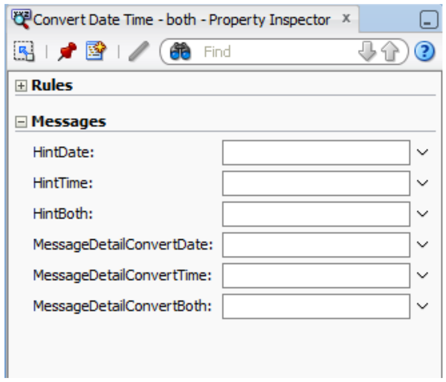
For example, entering a date that does not match the `dateStyle` attribute of the converter results in an error message, as shown in Figure 19–19.

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.

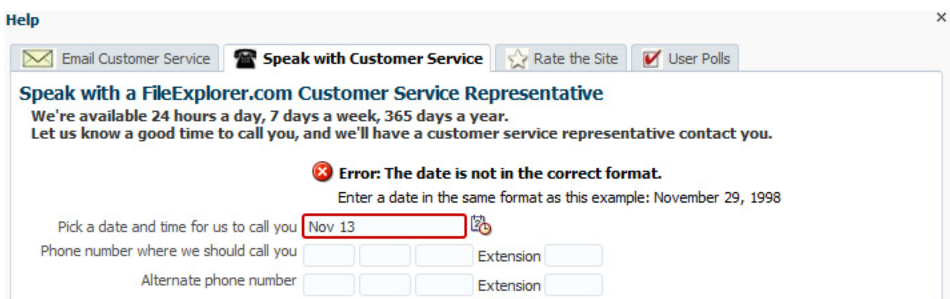
Figure 19–20 Message Attributes on a Converter



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:

- `<span>`
- `<b>`
- `<a>`
- `<i>`
- `<em>`
- `<br>`
- `<hr>`
- `<li>`
- `<ol>`
- `<ul>`
- `<p>`
- `<tt>`
- `<big>`
- `<small>`
- `<pre>`

19.3.1 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. For more information, see Section 19.3, "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. For more information, see Section 19.1.2, "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:

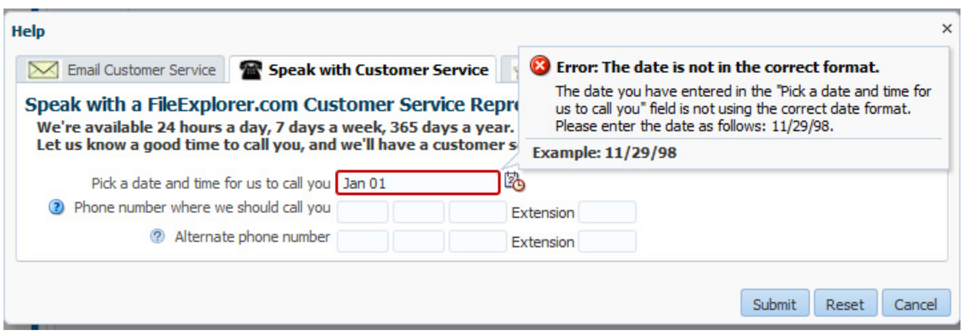
- If your application uses bidirectional or right-to-left display, do not start the message with the expected format parameter `{2}`, as it may not display correctly in Internet Explorer.
- You should try not to display the format parameter `{1}` in the message. Displaying the user input might be considered 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 more information about using resource bundles, see Chapter 32, "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. For more information, see Section 19.3.2, "How to Define Custom Validator and Converter Messages for All Instances of a Component."

19.3.2 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 Appendix B, "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 Section B.3.8, "af:validateLength."
2. Create or open a message bundle. For procedures how to create message bundles, see Section 32.3.1, "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 Section 32.3.5, "How to Register a Resource Bundle in Your Application." However, use the `<message-bundle>` tag, rather than the `<resource-bundle>` tag.

19.3.3 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. For more information, see Section 19.3, "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. For more information, see Section 19.1.2, "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.

19.3.4 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. For more information, see Section 19.3, "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. For more information, see Section 19.1.2, "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.

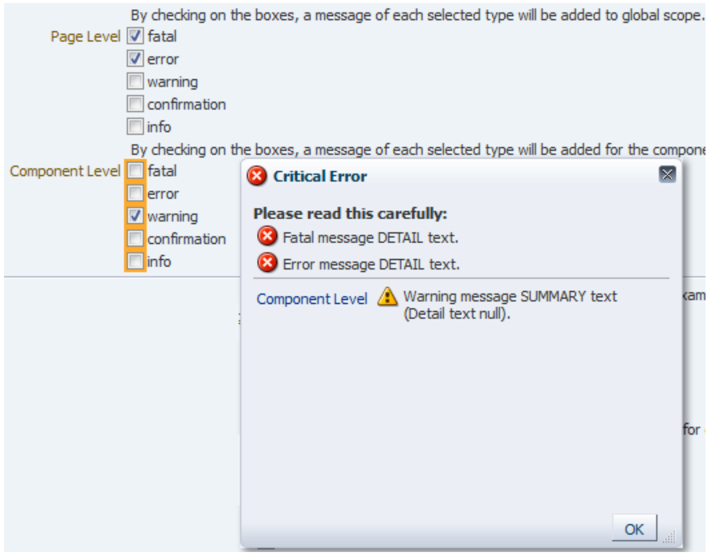
19.3.5 What Happens at Runtime: How Messages Are Displayed

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.

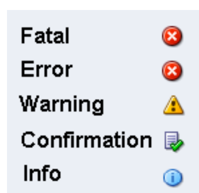
When conversion or validation fails on an `EditableValueHolder` ADF Faces component, `FacesMessages` objects are automatically added to the message queue on the `FacesContext` instance, passing in that component's ID. These messages are then displayed in the note window for the component. ADF Faces components 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:



19.4 Grouping Components with a Single Label and Message

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. Example 19–1 shows the JSF code for the `panelLabelAndMessage` component.

Example 19–1 panelLabelAndMessage Can Display a Single Label and Help Topic

```
<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

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

Use a `panelGroupLayout` component within a `panelLabelAndMessage` component to group the components for the required layout. For information about using the `panelGrouplayout` component, see Section 9.13, "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 Section 9.7, "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.

19.4.1 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. For more information, see Section 19.4, "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. For more information, see Section 19.1.2, "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:
   >
   > 1. Right-click the `panelLabelAndMessage` component in the Structure window.
   > 2. 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.

19.5 Displaying Help for Components

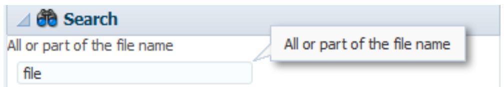
ADF Faces provides a framework that allows you to create and display three different types of help whose content comes from an external source, rather than as text configured on the component. Because it is not configured directly on the component, the content can be used by more than one component, 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

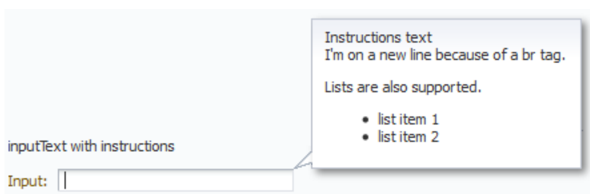
Instruction help displays help text in a note window when the focus is on the component. Figure 19–26 shows the help message in a note window.

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

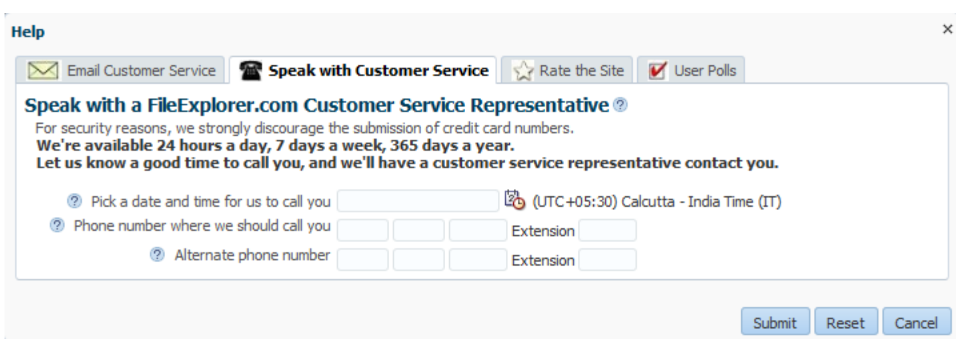


Table 19–1 shows the components that support Instruction help.

Table 19–1 Components That Support Instruction Help

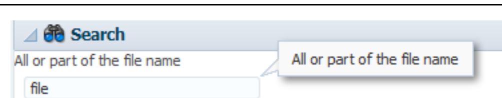
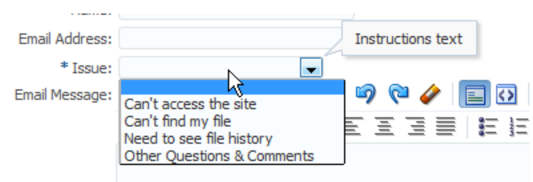
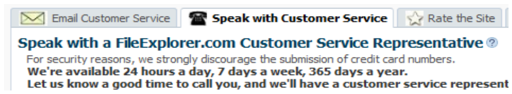
Supported Components	Help Placement	Example
Input components, Choose Color, Choose Date, Quick Query	Note window, on focus only	

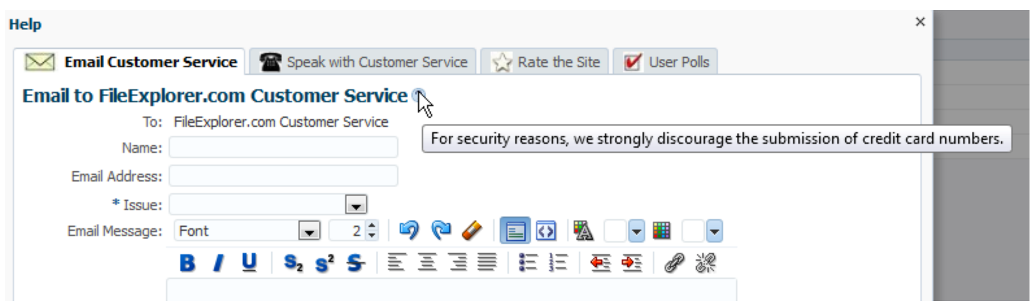
Table 19–1 (Cont.) Components That Support Instruction Help

Supported Components	Help Placement	Example
Select components	Note window, on hover and focus	
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). 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

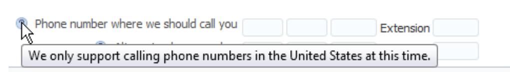
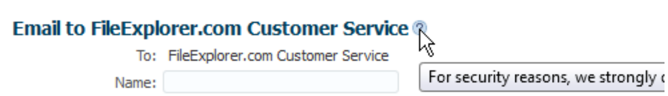
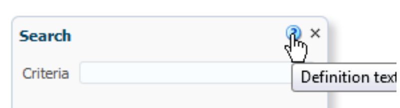
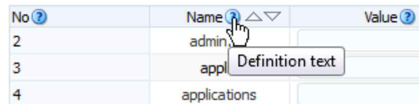
Supported Components	Help Icon Placement	Example
All input components, Select components, Choose Color, Choose Date, Query components	Before the label, or if no label exists, at the start of the field	
Panel Header, PanelBox, Show Detail Header	End of header text	
Panel Window, Dialog	Next to close icon in header	

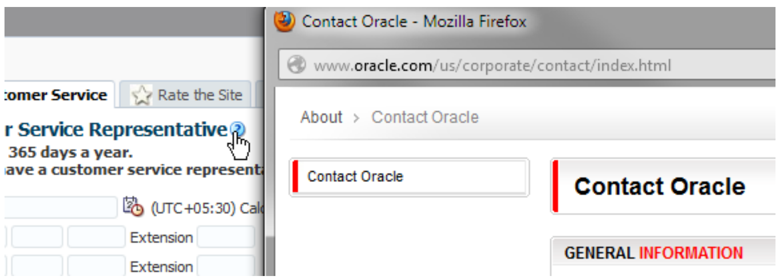
Table 19–2 (Cont.) Components That Support Definition Help

Supported Components	Help Icon Placement	Example
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 more information about using JavaScript to open the external help, rather than the help icon, see Section 19.5.4, "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 more information about creating help providers, see Section 19.5.1, "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. For more information, see Section 19.5.1, "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. For more information, see Section 19.5.1.4, "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. For more information, see Section 19.5.2, "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 name

   For more information, see Section 19.5.3, "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. For more information, see Section 19.5.4, "How to Use JavaScript to Launch an External Help Window."

19.5.1 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.

19.5.1.1 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. For more information, see Section 19.5, "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.

Example 19–2 shows an example resource bundle properties file with two topic IDs.

Example 19–2 Resource Bundle Help

```
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. For more information, see Section 19.5.2, "How to Register the Help Provider" and Section 19.5.3, "How to Access Help Content from a UI Component."

19.5.1.2 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. For more information, see Section 19.5, "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.

Example 19–3 shows an example of an XLIFF file that contains two topics.

Example 19–3 XLIFF Help

```
<?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. For more information, see Section 19.5.2, "How to Register the Help Provider" and Section 19.5.3, "How to Access Help Content from a UI Component."

19.5.1.3 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. For more information, see Section 19.5, "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 Example 19–4.

Example 19–4 Managed Bean that Returns a Map of Help Text Strings

```
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. For more information, see Section 19.5.1.4, "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. For more information, see Section 19.5.2, "How to Register the Help Provider" and Section 19.5.3, "How to Access Help Content from a UI Component."

19.5.1.4 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. For more information, see Section 19.5, "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.

   Example 19–5 shows the `DemoHelpProvider` class that extends the `ResourceBundleHelpProvider` class.

Example 19–5 Extending 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;
  }
}
```

Example 19–6 shows the `DemoELHelpProvider` class that extends the `ELHelpProvider` class.

Example 19–6 Extending 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;
  }
}
```

Example 19–5 and Example 19–6 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. Example 19–5 and Example 19–6 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. For more information, see Section 19.5.2, "How to Register the Help Provider" and Section 19.5.3, "How to Access Help Content from a UI Component."

19.5.1.5 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 more 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. For more information, see Section 19.5, "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.

19.5.2 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. For more information, see Section 19.5, "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 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 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 your are using external URL help, enter the fully qualified class name of the provider class you created.

Example 19–7 shows a resource bundle registered in the `adf-settings.xml` file.

Example 19–7 Registering a Resource Bundle as a Help Provider

```
<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>
```

Example 19–8 shows an XLIFF file registered in the `adf-settings.xml` file.

Example 19–8 Registering an XLIFF File as a Help Provider

```
<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>
```

Example 19–9 shows a bean registered in the `adf-settings.xml` file.

Example 19–9 Registering a Managed Bean as a Help Provider

```
<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>
```

Example 19–10 shows the external help provider registered in the `adf-settings.xml` file.

Example 19–10 Registering an External URL Help Provider

```
<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>
```

Example 19–11 shows an example of a custom help provider class registered in the `adf-settings.xml` file.

Example 19–11 Registering a Help Provider Class

```
<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>
```

19.5.3 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. For more information, see Section 19.5, "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 Example 19–3.

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.

19.5.4 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. For more information, see Section 19.5, "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.

   Example 19–12 shows the `launchHelp` function used to launch the `helpClient.jspx`.

Example 19–12 JavaScript to Launch an External Help Page

```
<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 more information about using the `clientListener` tag, see Section 4.4, "Listening for Client Events."

   Example 19–13 shows the code used to assign the `click` event of `button` component to launch the `helpClient.jspx` page.

Example 19–13 Page Code Used to Launch an External Help Window

```
<af:toolbar id="tb1">
  <af:button text="Launch help window" id="ctb1"
                         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>
```

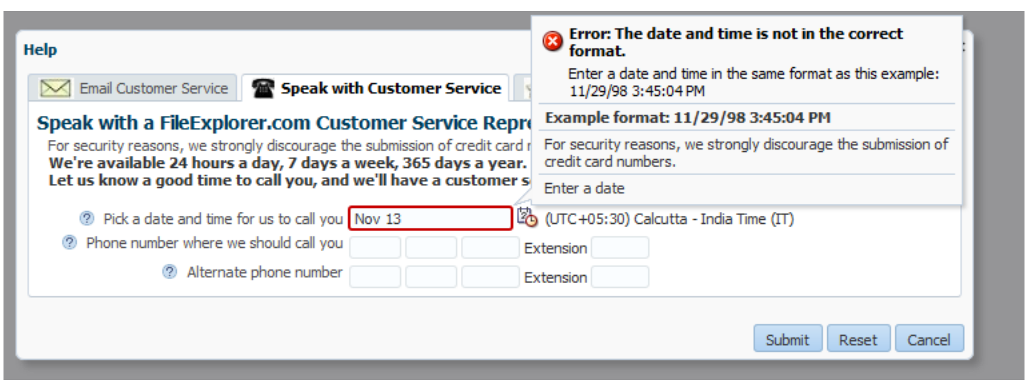
19.6 Combining Different Message Types

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 19–31 shows an `inputDate` component that contains a converter, instruction help, and a tip message.

Figure 19–31 Different Message Types Can Be Displayed at One Time



20

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:

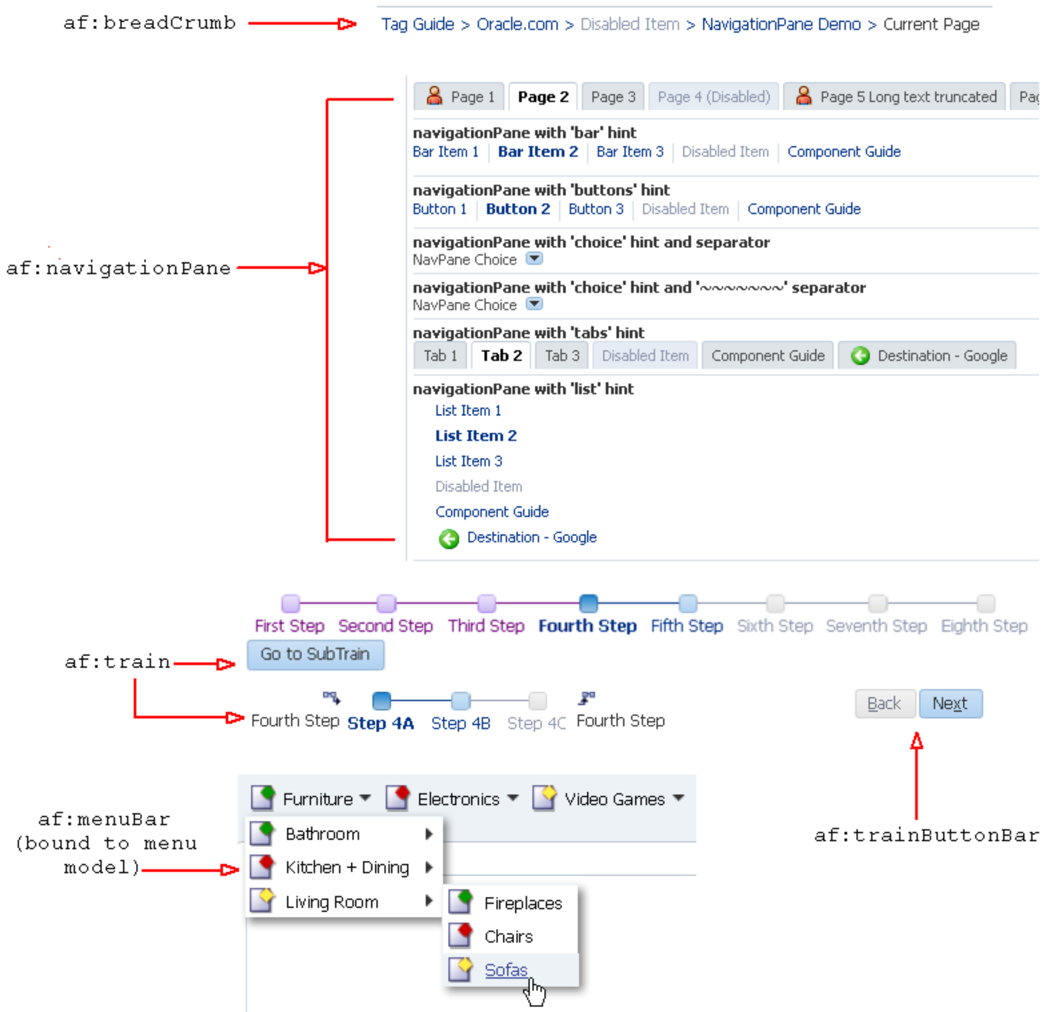
20.1 About Navigation Components

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.

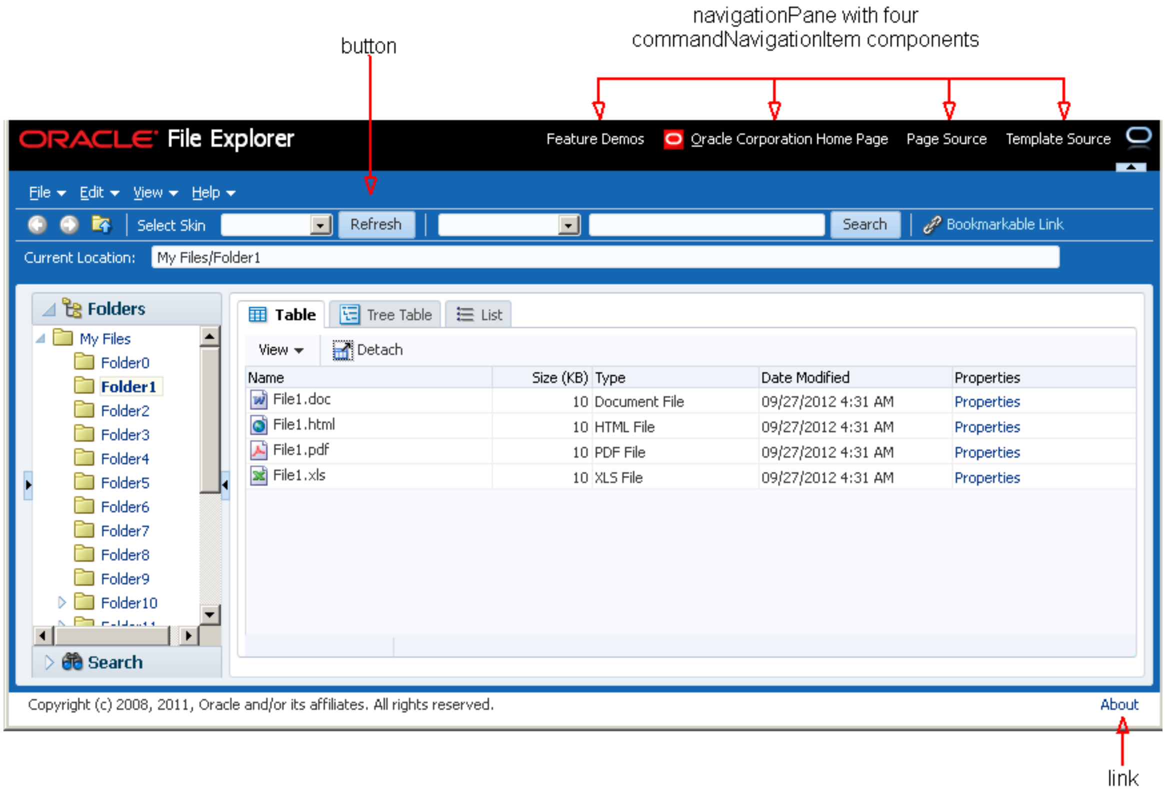
Figure 20–1 ADF Faces Navigation Components



20.1.1 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.

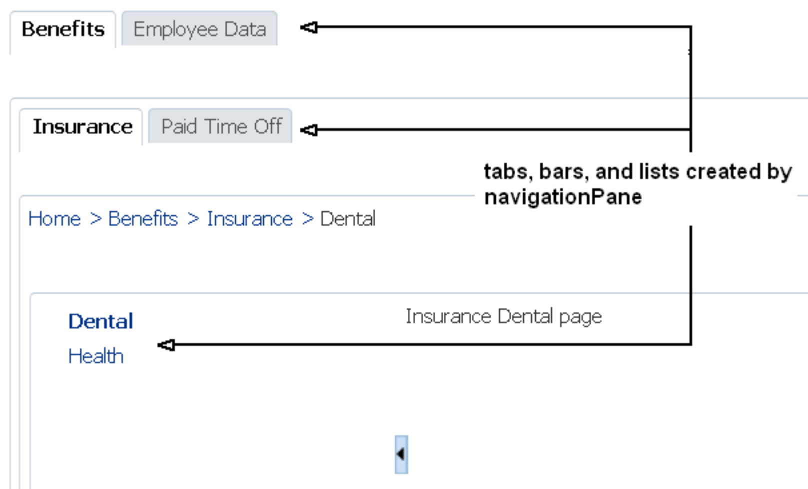
Figure 20–2 File Explorer Application Main Page



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.

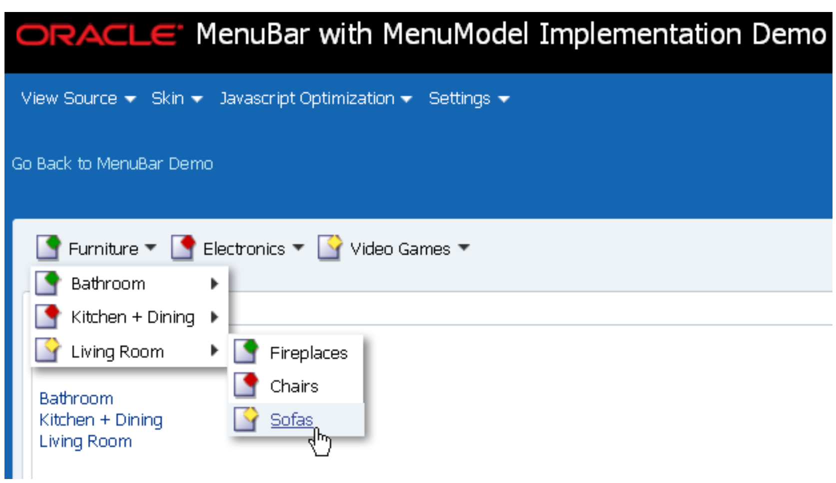
Figure 20–3 Page Showing Navigation Tab, Bar and List Links



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

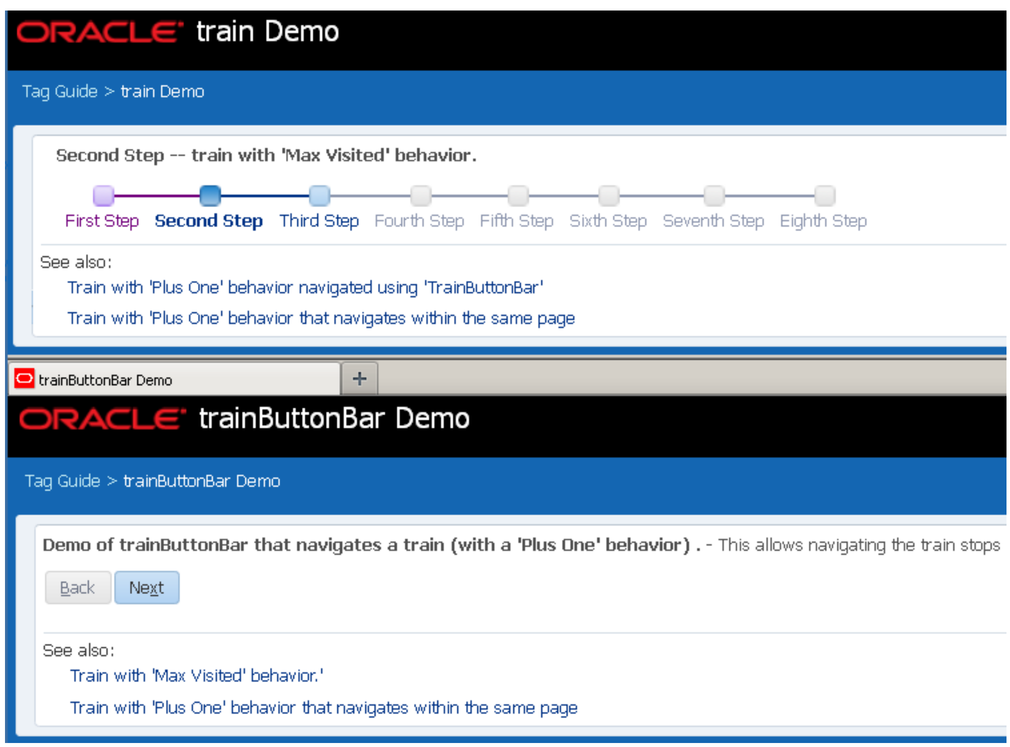


Whether you use `navigationPane` or `menuBar` (bound to a menu model) to create your page hierarchy, you can use the `breadCrumbs` component and a series of child `commandNavigationItem` components to provide users with a visual indication to their current location in the page hierarchy. As shown in 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 shown in Figure 20–6.

Figure 20–6 ADF Faces Train and TrainButtonBar Demonstration Pages



20.1.2 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. For more information, see Section 3.5.2, "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. For more information, see Chapter 6, "Handling Events."
- **Partial page rendering**: ADF Faces navigation components can be used to trigger partial rerendering of components on a page. For more information, see Chapter 8, "Rerendering Partial Page Content."
- **Accessibility**: You can make your navigation components accessible. For more information, see Chapter 33, "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. For more information, see Chapter 32, "Internationalizing and Localizing Pages."
- **Skins**: You can change the look and feel of navigation components by changing the skin. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

20.2 Common Functionality in Navigation Components

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 more information about the JSF lifecycle, see Chapter 5, "Using the JSF Lifecycle with ADF Faces." Also note that navigation in an ADF Faces application may use partial page rendering. For more information, see Chapter 8, "Rerendering Partial Page Content."

20.3 Using Buttons and Links for Navigation

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. For more information, see Chapter 16, "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

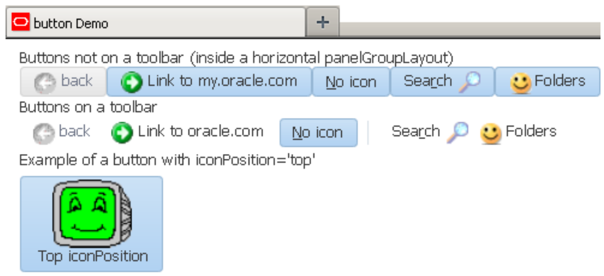
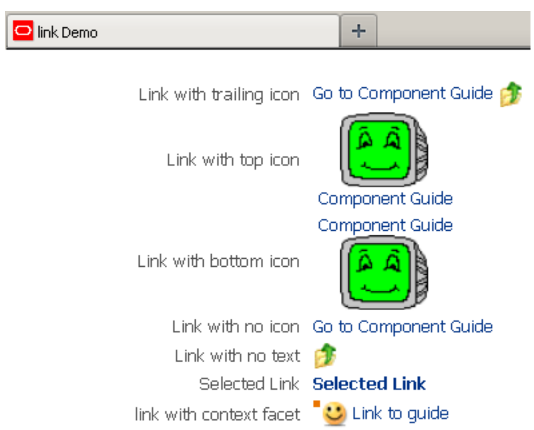


Figure 20–8 shows a number of the options that you can configure for the link component.

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. For more information, see Section 16.3, "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 more information about email and print output modes, see Chapter 37, "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). For more information, see Section 20.4, "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 Section 9.2.5, "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`, as described in the "How to Enable Implicit Save Points" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

20.3.1 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. For more information, see Section 20.3, "Using Buttons and Links for Navigation."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 20.1.2, "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 Section 33.3.4, "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 more information about configuring the navigation between pages, see Section 3.3, "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 more information about using managed bean methods to open dialogs, see Chapter 15, "Using Popup Dialogs, Menus, and Windows." For more information about outcome strings and navigation in JSF applications, see the Java EE 6 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. For more information, see Section 8.2, "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. For more information, see Section 5.2, "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 Section 9.2.5, "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 the "Using the ADF Faces Dialog Framework Instead of Bounded Task Flows" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

To use buttons and links to invoke popups without writing any JavaScript code, see Section 15.3, "Declaratively Invoking a Popup."

20.3.2 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. For more information, see Section 20.3, "Using Buttons and Links for Navigation."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 20.1.2, "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 Section 33.3.4, "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.

20.3.3 What You May Need to Know About Using Partial Page Navigation

As described in Section 8.5, "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.

20.4 Configuring a Browser's Context Menu for Links

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.

20.4.1 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. For more information, see Section 20.4, "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. For more information, see Section 20.1.2, "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.

20.4.2 What Happens When You Configure a Browser's Context Menu for Command Links

If you followed the procedure outlined in Section 20.4.1, "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.

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>
```

For more information about ADF Faces configuration options in your application's `web.xml` file, see Section A.2, "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 Section 20.4, "Configuring a Browser's Context Menu for Links."

20.5 Using Buttons or Links to Invoke Functionality

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:

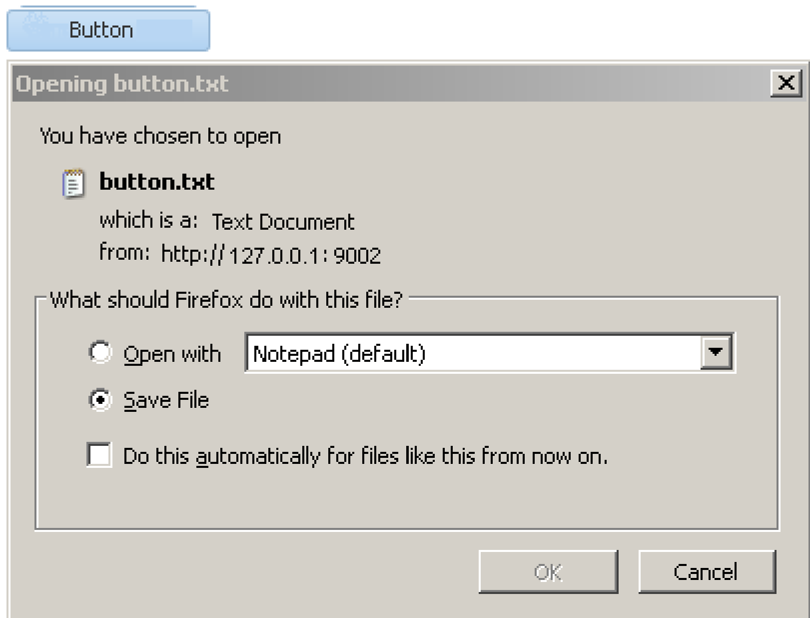
- `exportCollectionActionListener`: Use to export data from the `table`, `tree` and `treeTable` components to an Excel spreadsheet. For more information, see Section 12.12, "Exporting Data from Table, Tree, or Tree Table."
- `fileDownloadActionListener`: Use to initiate a file download from the server to the local hard drive. For more information, see Section 20.5.1, "How to Use an Action Component to Download Files."
- `resetListener`: Use to reset submitted values. However, no data model states will be altered. For more information, see Section 20.5.2, "How to Use an Action Component to Reset Input Fields." If the input components render in a popup, see Section 15.7, "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. For more information, see Section 11.2.3, "How to Add a Button to Reset the Form."

20.5.1 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 Section 11.9, "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.

Example 20–2 shows the tags of a button with the `fileDownloadActionListener` tag to download the file named `hello.txt` to the user.

Example 20–2 File Download Using Button and fileDownloadActionListener Tag

```
<af:button value="Say Hello">
  <af:fileDownloadActionListener filename="hello.txt"
      contentType="text/plain; charset=utf-8"
      method="#{bean.sayHello}"/>
</af:button>
```

Example 20–3 shows a managed bean method used to process the file download.

Example 20–3 Managed Bean Method Used to Process 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
`oracle.adf.view.rich.portal.FILE_DOWNLOAD_MAX_MEM`	Integer	Specify the maximum size in kilobytes of the file that the `fileDownloadActionListener` 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 `FILE_DOWNLOAD_TEMP_DIR`. If you do not specify a value for this parameter in the `web.xml` file, it defaults to `100` kilobytes.
`oracle.adf.view.rich.portal.FILE_DOWNLOAD_MAX_DISK_SPACE`	Integer	Specify the maximum size in kilobytes of the file that the `fileDownloadActionListener` 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 `web.xml` file, it defaults to 2000.
`oracle.adf.view.rich.portal.FILE_DOWNLOAD_TEMP_DIR`	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 `java.io.tempDir`.

For more information about configuring your `web.xml` file, Section A.2, "Configuration in web.xml." For information about how to create a JSF portlet, see the *Oracle Fusion Middleware Developer's Guide for Oracle WebCenter*.

Before you begin:

It may help to understand how a component's attributes affect functionality. For more information, see Section 20.5, "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. For more information, see Section 20.1.2, "Additional Functionality for Navigation Components."

You will need to complete this task:

> Create an action component, as described in Section 20.3, "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>
```

20.5.2 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. For more information, see Section 11.2.3, "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 more information about this use case, see Section 15.7, "Resetting Input Fields in a Popup."

Before you begin:

It may help to understand how an action component's attributes affect functionality. For more information, see Section 20.5, "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. For more information, see Section 20.1.2, "Additional Functionality for Navigation Components."

You will need to complete this task:

Create an action component, as described in Section 20.3, "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.

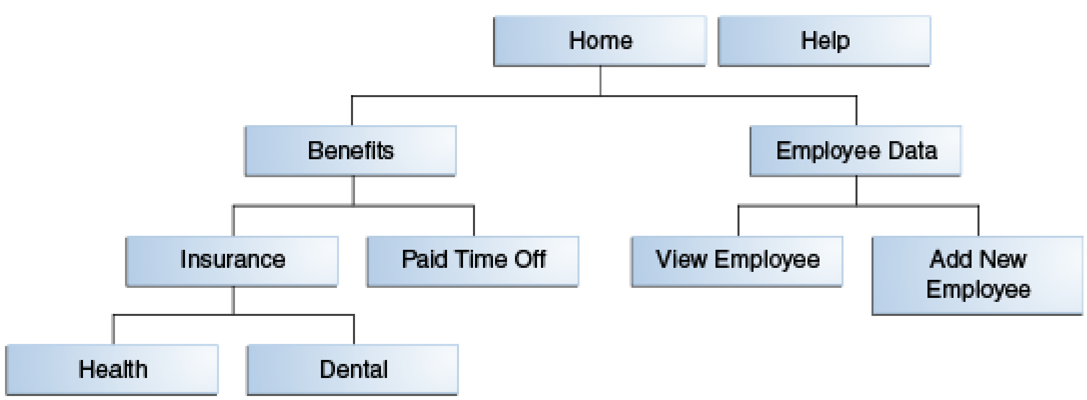
   Click **Help** in the Insert Reset Listener dialog to view a complete list of supported values.

20.6 Using Navigation Items for a Page Hierarchy

> **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. For details, see the "Creating a Page Hierarchy Using Task Flows" section 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. For information about using themes and skins, see Chapter 31, "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 Section 10.3, "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 Chapter 16, "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 more information about creating a navigational hierarchy using the `XMLMenuModel`, see Section 20.7, "Using a Menu Model to Create a Page Hierarchy". For more information about manually creating a navigational hierarchy, see Section 20.8, "Creating a Simple Navigational Hierarchy".

20.6.1 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. For more information, see Section 20.6, "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. For more information, see Section 20.1.2, "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 20–4.

   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.

Example 20–4 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>
```

For more information about creating navigation cases in JDeveloper, see Section 3.3, "Defining Page Flows."

20.7 Using a Menu Model to Create a Page Hierarchy

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*.

Section 20.6, "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 more information about a tree model, see Section 12.6, "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 Section 9.10, "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 Section 20.6.1, "How to Create Navigation Cases for a Page Hierarchy."
- Create the `XMLMenuModel` metadata. See Section 20.7.1, "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 Section 20.7.2, "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 Chapter 10, "Creating and Reusing Fragments, Page Templates, and Components."

- On each page, bind the `navigationPane` and `breadCrumbs` components to the `XMLMenuModel` class. See Section 20.7.3, "How to Bind the navigationPane Component to the Menu Model" and Section 20.7.4, "How to Use the breadCrumbs

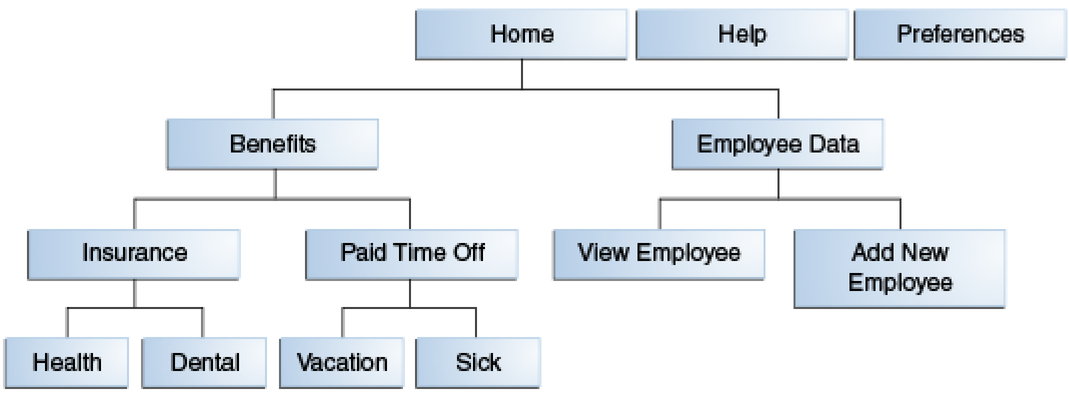
Component with a Menu Model." To bind the `menuBar` component, see Section 20.7.5, "How to Use the menuBar Component with a Menu Model."

20.7.1 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.

Example 20–5 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.

Example 20–5 XMLMenuModel Metadata File Sample 1

```
<?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 Example 20–5 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 Example 20–6) so that the menu model could be reused across an application.

Example 20–6 Benefits XMLMenuModel Metadata File

```
<?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.

Example 20–7 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.

Example 20–7 XMLMenuModel Metadata File Sample 2

```
<?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. For more information, see Section 20.7, "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. For more information, see Section 20.1.2, "Additional Functionality for Navigation Components."

To create the XMLMenuModel metadata:

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**.

   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 the "Creating a Page Hierarchy Using Task Flows" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

   ---

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 Example 20–8.

Example 20–8 Blank XMLMenuModel Metadata File

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"></menu>
```

   For information about the managed bean configuration that JDeveloper automatically adds for you in `faces-config.xml`, see Section 20.7.2, "What Happens When You Use the Create ADF Menu Model Wizard."

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
`resourceBundle`	Optional. This is the resource bundle to use for the labels (visible text) of the navigation items at runtime. For example, `org.apache.myfaces.demo.xmlmenuDemo.resource.MenuBundle`.
`var`	If using a resource bundle, specify an ID to use to reference the bundle in EL expressions for navigation item labels. For example, `#{bundle.somelabel}`. See Example 20–9 for a sample `XMLMenuModel` metadata file that uses a resource bundle.
`xmlns`	Required. Set to `http://myfaces.apache.org/trinidad/menu`

   Example 20–9 shows sample `XMLMenuModel` metadata code that uses EL expressions to access a resource bundle for the navigation item labels.

Example 20–9 XMLMenuModel Using Resource Bundle

```
<?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>
```

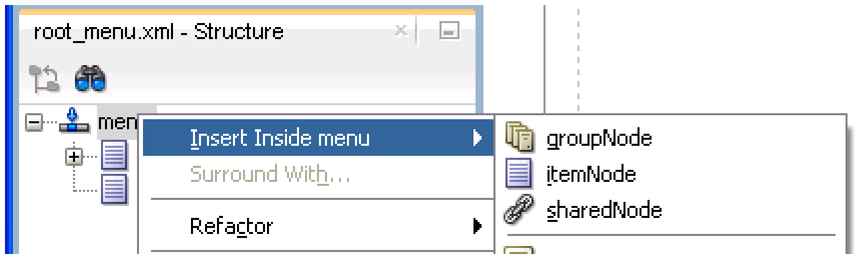
> **Note:** When you use a `sharedNode` element to create a submenu and you use resource bundles for the navigation item labels, it is possible that the shared menu model will use the same value for the `var` attribute on the root `menu` element. The `XMLMenuModel` class handles this possibility during parsing by ensuring that each resource bundle is assigned a unique hash key.

   For more information about using resource bundles, see Chapter 32, "Internationalizing and Localizing Pages."

5. 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.

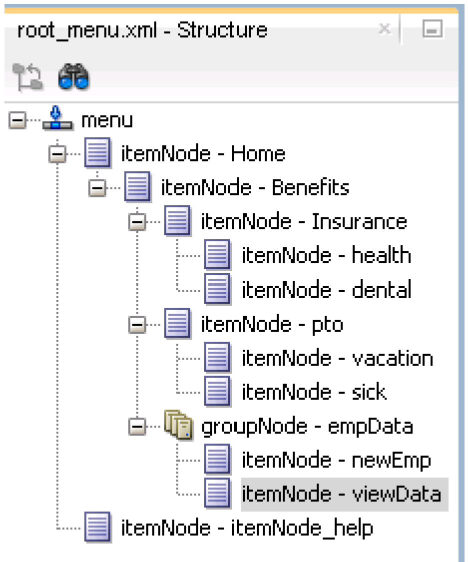
  You can insert a `sharedNode` element anywhere within the hierarchy. For example, in the code shown in Example 20–10, the `sharedNode` element adds a submenu on the same level as the first-level Employee Data node.

Example 20–10 SharedNode Sample Code

```
<?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
`action`	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 `from-outcome` value to the navigation case for that node as configured in the `faces-config.xml` file.
`destination`	Specify the URI of the page to navigate to when the node is selected, for example, `http://www.oracle.com`. If the destination is a JSF page, the URI must begin with "`/faces`". Alternatively, specify an EL method expression that evaluates to the URI. If both `action` and `destination` are specified, `destination` takes precedence over `action`.
`focusViewId`	Required. The URI of the page that matches the node's navigational result, that is, the `to-view-id` value of the navigation case for that node as specified in the `faces-config.xml` file. For example, if the action outcome of the node navigates to `/page_one.jsf` (as configured in the `faces-config.xml` file), then `focusViewId` must also be `/page_one.jsf`. The `focusViewId` does not perform navigation. Page navigation is the job of the `action` or `destination` attributes. The `focusViewId`, however, is required for the `XMLMenuModel` to determine the correct focus path.
`id`	Required. Specify a unique identifier for the node. As shown in Example 20–5, it is good practice to use "inX" for the ID of each `itemNode`, where for example, "inX" could be in1, in11, in111, in2, in21, in 211, and so on.

Table 20–3 (Cont.) itemNode Element Attributes

Attribute	Description
`label`	Specify the label text to display for the node. Can be an EL expression to a string in a resource bundle, for example, `#{bundle.somelabel}`, where `bundle` must match the root `menu` element's `var` attribute value.

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 Example 20–11. 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.

Example 20–11 groupNode Elements

```
<?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
`id`	A unique identifier for the group node. As shown in Example 20–11, it is good practice to use `gnX` for the ID of each `groupNode`, where for example, `gnX` could be `gn1`, `gn2`, and so on.

Table 20–4 (Cont.) GroupNode Element Attribute

Attribute	Description
idref	Specify the ID of a child node, which can be an itemNode, or another groupNode. When adding a groupNode as a child node, that child in turn can reference another groupNode and so on, but eventually an itemNode child must be referenced as the last child. The idref attribute can contain more than one child ID, separated by spaces; the IDs are processed in the order they are listed.
label	Specify the label text to display for the group node. Can be an EL expression to a string in a resource bundle, for example, #{bundle.somelabel}.

Table 20–5 sharedNode Element Attribute

Attribute	Description
ref	Specify the managed bean name of another XMLMenuModel class, as configured in the faces-config.xml file, for example, #{shared_menu}. At runtime, the referenced navigation menu is created, inserted as a submenu into the main (root) menu, and rendered.

20.7.2 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.

Example 20–12 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.

Example 20–12 Managed Bean Configuration for XMLMenuModel in faces-config.xml

```
<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 (as shown in Example 20–10), 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 Example 20–13.

Example 20–13 Managed Bean for Shared Menu Model in faces-config.xml

```
<!-- 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.

20.7.3 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 as shown in Example 20–5), 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 and Example 20–5, 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 Section 20.8.1, "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 Example 20–14.

Example 20–14 navigationPane Component Bound to XMLMenuModel Managed Bean

```
<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 by its ADF skin when you specify the theme value for the component. You can change how the themes display. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. For more information, see Section 20.7, "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. For more information, see Section 20.1.2, "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 as shown in Example 20–5), 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 20–15 shows the JSF code for binding to a menu model that has four levels of hierarchical nodes.

Example 20–15 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 Section 20.8.3, "What You May Need to Know About Removing Navigation Tabs."

20.7.4 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. For more information, see Section 20.7, "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. For more information, see Section 20.1.2, "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 (as shown in Example 20–10), 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.

Example 20–16   breadCrumbs Component Bound to the XMLMenuModel

```
<af:breadCrumbs var="menuNode" value="#{root_menu}">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
                              action="#{menuNode.doAction}"/>
  </f:facet>
</af:breadCrumbs>
```

20.7.5 How to Use the menuBar Component with a Menu Model

As described in Chapter 16, "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 by its ADF skin when you specify the theme value for the component. You can change how the themes display. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

Before you begin:

It may help to understand how the attributes of navigation components affect functionality. For more information, see Section 20.7, "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. For more information, see Section 20.1.2, "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 (as shown in Example 20–10), 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.

Example 20–17 menuBar Component Bound to the XMLMenuModel

```
<af:menuBar var="menuNode" value="#{root_menu}">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
                              action="#{menuNode.doAction}"/>
  </f:facet>
</af:menuBar>
```

20.7.6 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.

20.7.7 What You May Need to Know About Using Custom Attributes

Custom attributes that you have created can be displayed, but only for itemNode elements. To add an itemNode element to access the value of a custom attribute, you need to get the tree from the menu model by:

- Calling the menu models getWrappedData() method
- Calling the getFocusRowKey() method to get the current focus path
- Using this focus path to traverse the tree and return a list of nodes in the focus path
- Testing one or more of these nodes for custom attribute(s) by calling the getCustomProperty() API

Example 20–18 shows an example of the required code.

Example 20–18 Accessing Custom Attributes from the XMLMenuModel

```
/**
  * Returns the nodes corresponding to a focus path
  *
  * @param tree
  * @param focusPath
  */
  public List getNodesFromFocusPath(TreeModel tree, ArrayList focusPath)
  {
    if (focusPath == null || focusPath.size() == 0)
      return null;

    // Clone the focusPath cause we remove elements
    ArrayList fp = (ArrayList) focusPath.clone();

    // List of nodes to return
    List nodeList = new ArrayList<Object>(fp.size());

    // Convert String rowkey to int and point to the
    // node (row) corresponding to this index
    int targetNodeIdx = Integer.parseInt((String)fp.get(0));
    tree.setRowIndex(targetNodeIdx);

    // Get the node
    Object node = tree.getRowData()

    // put the Node in the List
    nodeList.add(node);

    // Remove the 0th rowkey from the focus path
    // leaving the remaining focus path
    fp.remove(0);

    // traverse into children
    if (   fp.size() > 0
        && tree.isContainer()
        && !tree.isContainerEmpty()
```

```
          )
        {
          tree.enterContainer();

          // get list of nodes in remaining focusPath
          List childList = getNodesFromFocusPath(tree, fp);

          // Add this list to the nodeList
          nodeList.addAll(childList);

          tree.exitContainer();
        }

        return nodeList;
      }

    public String getElementLabel(XMLMenuModel model, Object myVal, String myProp)
    {
       TreeModel tree = model.getWrappedData();

       Object node = findNodeByPropertyValue(tree, myVal, myProp);

       FacesContext context = FacesContext.getCurrentInstance();
       PropertyResolver resolver = context.getApplication().getPropertyResolver();

       String label = (String) resolver.getValue(node, _LABEL_ATTR);

       return label;
    }

    public Object findNodeByPropertyValue(TreeModel tree, Object myVal, String
  myProp)
    {
      FacesContext context = FacesContext.getCurrentInstance();
      PropertyResolver resolver = context.getApplication().getPropertyResolver();

      for ( int i = 0; i < tree.getRowCount(); i++)
      {
        tree.setRowIndex(i);

        // Get a node
        Object node = tree.getRowData();

        // Get the value of the attribute of the node
        Obect propVal = resolver.getValue(node, myProp);

        if (propVal == myVal)
        {
          return node;
        }

        if (tree.isContainer() && !tree.isContainerEmpty())
        {
          tree.enterContainer();
          node = findNodeByPropertyValue(tree, myVal, myProp);

          if (node != null)
            return node;

          tree.exitContainer();
```

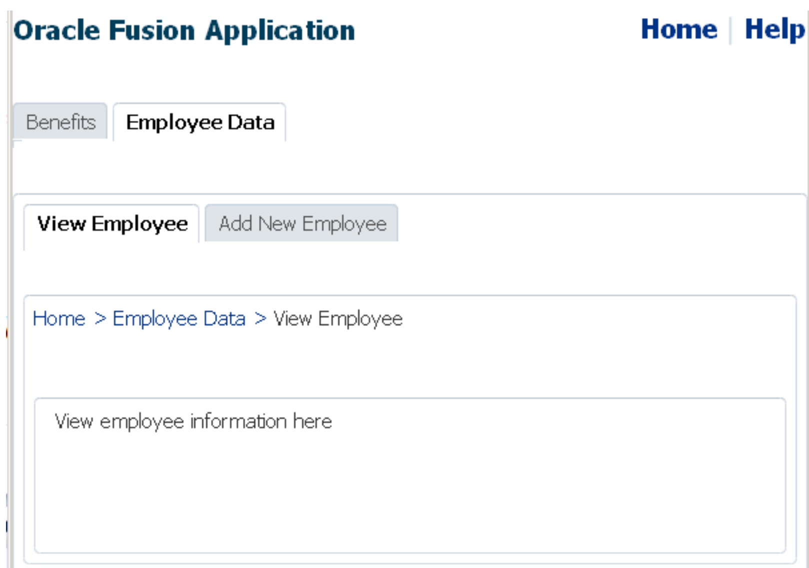
```
      }guap
    }
    return null;
  }
```

20.8 Creating a Simple Navigational Hierarchy

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 Section 20.7, "Using a Menu Model to Create a Page Hierarchy".

Section 20.6, "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.

20.8.1 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. For more information, see Section 20.8, "Creating a Simple Navigational Hierarchy."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 20.1.2, "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 Example 20–19.

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.

Example 20–19 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>
```

For more information about creating navigation cases in JDeveloper, see Section 3.3, "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 by its ADF skin when you specify the theme value for the component. You can change how the themes display. For more information, see Chapter 31, "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 Section 10.3, "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 20–20 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"`.

Example 20–20 Sample Code Using Individual Navigation Items on One Page

```
<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 20–21.

Example 20–21 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);
}
```

20.8.2 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. For more information, see Section 20.8, "Creating a Simple Navigational Hierarchy."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 20.1.2, "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 Section 20.8.1, "How to Create a Simple Page Hierarchy." If referencing a method, that method must return the required string.

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 20–22.

Example 20–22 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>
```

20.8.3 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). For more information about events, see Chapter 6, "Handling Events." For information about using popup dialogs and windows, see Chapter 15, "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 20–23.

Example 20–23   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 ...
    }
  }
}
```

20.8.4 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 ellipses 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.

20.8.5 What You May Need to Know About Navigation Tabs in a Compressed Layout

Built-in overflow indicators appear if the application window is in a compressed layout. That is, the application window is not wide enough to display all the navigation items. These overflow indicators render dropdown lists where the user can choose the navigation item to navigate to, as shown in Figure 20–18.

Figure 20–18 Overflow Indicator for a navigationPane Component in Compressed Layout

Rather than display overflow indicators (the default behavior), 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 items that are not currently visible. Configuring the `-tr-layout-type` skinning key also renders all navigation items 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 its default value (`overflow`).

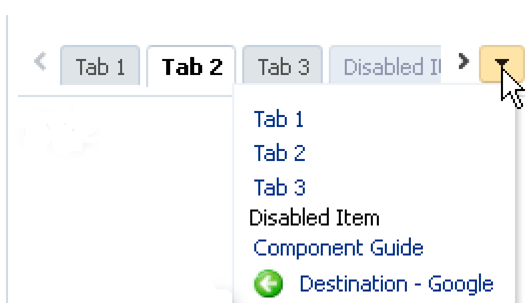
Example 20–24 shows how you configure the `-tr-layout-type` skinning key in your application's ADF skin file to render a conveyor belt for the `navigationPane` and `panelTabbed` components. For more information about skinning, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

Example 20–24 Configuring the -tr-layout-type Skinning Key to Render a Conveyor Belt

```
af|panelTabbed,
af|navigationPane {
  -tr-layout-type: conveyor;
}
```

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



20.9 Using Train Components to Create Navigation Items for a Multistep Process

---

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. For details, see the "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

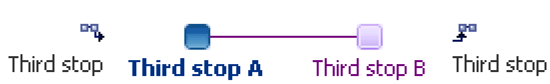
The `train` component renders each configured step represented as a *train stop*, and with all the stops connected by lines. Each train stop has an image (for example, a square block) with a label underneath the image.

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 **Back** and **Next** buttons, as shown in Figure 20–22. These **Back** and **Next** buttons 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 **Back** and **Next** buttons without displaying the stops when not all of the stops will be visited based on some conditional logic.

Figure 20–22 Navigation Buttons Rendered by a trainButtonBar Component

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. For more information, 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 (For more information, see Chapter 12, "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 Section 20.7.3, "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. Example 20–25 shows the simplified binding for a train.

> **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. For more information, see Section 20.9.3, "How to Bind to the Train Model in JSF Pages."

Example 20–25 Simplified Train Model Binding

```
<af:train value="#{simpleTrainModel}"/>
```

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 Section 20.8.1, "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 more information about the `redirect` element, see the JavaServer Faces specification. For more information about partial page rendering, see Chapter 8, "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 Section 20.9.1, "How to Create the Train Model."
- Configure managed beans for the train model. See Section 20.9.2, "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 Section 20.9.3, "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.

20.9.1 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 Demonstration application.

Before you begin:

It may help to understand how a train component's attributes affect functionality. For more information, see Section 20.9, "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. For more information, see Section 20.1.2, "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, as shown in Example 20–25.

   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. For more information, 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. For more information, 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 more information about messages, see Chapter 19, "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. For more information, see Step 2.

2. Create a class based on the `MenuModel` class to facilitate the construction of a train model.

   The `MenuModel` implementation of your train model must have a specific train behavior. The `ProcessMenuModel` class in the `org.apache.myfaces.trinidad.model` package is a reference implementation of the `MenuModel` 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 `ProcessMenuModel` class, or create your own.

   In your train model class, you override the `getFocusRowKey()` method (see the `MenuModel` class) and implement a train behavior (see the `ProcessMenuModel` and `ProcessUtils` classes).

   The train behaviors provided in the `ProcessMenuModel` 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 `ProcessMenuModel` 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 more information about how the `immediate` attribute works, see Section 5.2, "Using the Immediate Attribute."

The `ProcessMenuModel` 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. For more information, see the "*`pageName`*`PageDef.xml`" section 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 (For more information, see Section 20.9.2, "How to Configure Managed Beans for the Train Model"). If the `maxPathKey` value is `null`, then ADF Faces uses the Plus One behavior.

20.9.2 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. For more information, see Section 20.9, "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. For more information, see Section 20.1.2, "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 Section 20.9.1, "How to Create the Train Model." Example 20–26 shows sample managed bean code for train stops in the `faces-config.xml` file.

Example 20–26 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 -->
...
```

The managed properties set the values to the train stop model object (the class created in Step 1 in Section 20.9.1, "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 20–30).

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 20–27.

Example 20–27 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>
```

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 20–28 shows sample managed bean code for creating the train stop list.

Example 20–28 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>
```

   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 20–27).

Example 20–29 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>
```

   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 20–30 shows sample managed bean code for a train model.

Example 20–30 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>
```

   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.

20.9.3 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. For more information, see Section 20.9, "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. For more information, see Section 20.1.2, "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 20–31.

Example 20–31 Simple Implementation of a Train Component Bound to a Menu Model

```
<af:train value="#{trainMenuModel}"/>
<af:trainButtonBar value="#{trainMenuModel}"/>
```

The `trainMenuModel` EL expression is the managed bean name for the train model (see Example 20–30).

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 20–32.

Example 20–32 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>
```

21

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. For more information, see the "Creating a Basic Databound Page" and "Creating ADF Databound Tables" chapters of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter incudes the following sections:

21.1 About Determining Components 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:

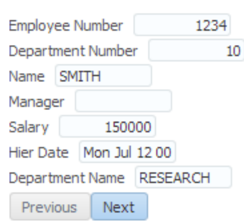
- `inputText`

- `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 the 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



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 Example 21–1.

Example 21–1 Dynamic Component Used in a Form

```
<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 (for more information, see Chapter 12, "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.

Example 21–2 shows a dynamic component used in a table.

Example 21–2 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 Example 21–3

Example 21–3 Dynamic Component Used in Groups

```
<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.

21.2 Creating the Model for a Dynamic Component

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.

21.2.1 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. For more information, see Section 21.1, "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

   Example 21–4 shows how you might create an attribute definition, given the metadata created in Step 1.

Example 21–4 Create an Attribute Definition

```
public void addAttributeDef (String name, String label, Class dataType) {
  TestAttributeDef attributeDef = new TestAttributeDef(name, label, dataType);
    _attributes.put(name, attributeDef);
 }
```

Example 21–5 then shows how to return a list of Employee object attributes with the attribute definitions populated.

Example 21–5 Define the Attributes

```
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. Example 21–6 shows how the Name and DataType might be returned.

Example 21–6 Access the Attribute Definitions

```
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.

Example 21–7 Create an AttributesModel

```
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. Example 21–8 shows the code for returning the `AttributesModel`, as well as the logic for accessing next and previous records using Action events.

Example 21–8 Managed Bean Code for a Dynamic Component

```
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.

21.2.2 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. For more information, see Section 21.1, "About Determining Components at Runtime."

To create the model using groups:

1. Create the attribute definition class, as described in Step 1 of Section 21.2.1, "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`. Example 21–9 shows how the get method might be coded.

Example 21–9 Get Method for a Category Uses groupName

```
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. Example 21–10 shows a `GroupAttributeDescriptor` class.

Example 21–10 Class for the GroupAttributeDescriptor Object

```
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. Example 21–11 shows how you might code that logic.

Example 21–11 Logic to Add Groups to the hierarchicalAttributes

```
 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.

21.3 Adding a Dynamic Component as a Form to a Page

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.

21.3.1 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. For more information, see Section 21.1, "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 Section 9.7, "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 Section 20.3.1, "How to Use Buttons and Links for Navigation and Deliver ActionEvents."

   Example 21–12 shows buttons whose `actionListener` attributes are bound to logic on a managed bean that navigates between the records in the model.

Example 21–12   Using Buttons to Navigate Between Records in an AttributesModel

```
<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>
```

Example 21–13 shows the corresponding managed bean code.

Example 21–13   Managed Bean Code to Access the Next Record for a Dynamic Component

```
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));
}
```

21.3.2 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. For more information, see Section 21.1, "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 Section 9.7, "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]}
      ```

21.4 Adding a Dynamic Component as a Table to a Page

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.

21.4.1 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. For more information, see Section 21.1, "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**, and bind it to the data for your table.

   For more information about tables, see Section 12.3, "Displaying Data in Tables."

2. In the Structure window, delete the column component and its contents that were created for you. You will manually add a column in a later step.

3. In the Components window, from the Operations panel, drag and drop an **Iterator** as a child to the `table` component.

4. 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: `col`

5. From the Components window, drag and drop a **Column** as a child to the iterator component. In the Properties window, bind **HeaderText** to the `label` property of the attributes returned by the iterator, for example, `#{col.label}`.

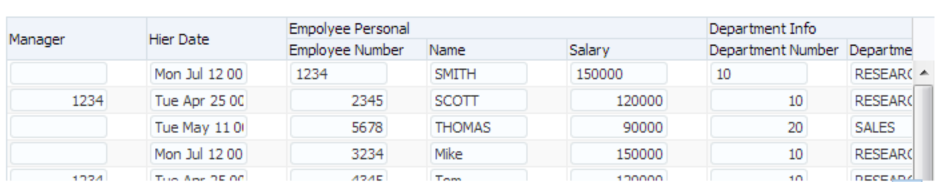
6. From the Components window, drag and drop a **Dynamic Component** as a child to the `column` component.

7. In the Properties Window, set the following
   - **AttributeModel**: An EL expression that resolves to the variable created for the iterator, for example: `#{col}`.
   - **Value**: An EL expression that resolves to each attribute name on the `AttributesModel` object. Because this needs to be stamped out for every row, you need to bind this to the value returned by the table component, for example, `#{row[col.name]}`.

21.4.2 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	Empolyee Personal			Department Info	
		Employee Number	Name	Salary	Department Number	Departme
	Mon Jul 12 00	1234	SMITH	150000	10	RESEAR(
1234	Tue Apr 25 0C	2345	SCOTT	120000	10	RESEAR(
	Tue May 11 0	5678	THOMAS	90000	20	SALES
	Mon Jul 12 00	3234	Mike	150000	10	RESEAR(

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. For more information, see Section 21.1, "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**. In the Create ADF Faces Table dialog, select **Bind Data Now**. Click **Browse** for the **Table Data Collection** to open the Select Table Data Collection dialog. Select the values property on the managed bean you created for the model.

   For more information about tables, see Section 12.3, "Displaying Data in Tables."

2. In the Structure window, delete the column component and its contents that were created for you. You will manually add a column in a later step.

3. In the Components window, from the Operations panel, drag and drop an **Iterator** as a child to the table component.

4. 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.heirarchicalAttributes}"
     ```

   - **Var**: A String that can be used to access the attributes on the model, for example: column

5. From the Components window, drag and drop a **Column** as a child to the iterator component. In the Properties window, bind **HeaderText** to the label property of the groups returned by the iterator, for example, #{column.label}.

6. Drag and drop a **Switcher** as a child to the column component.

7. In the Properties window, set the following:

   - **FacetName**: An EL expression that resolves to the descriptorType property of the values returned by the variable on the parent iterator. For example:

     ```
     #{column.descriptorType}
     ```

   - **DefaultFacet**: Name the facet that will hold the attributes that don't belong to a group, for example, ATTRIBUTE.

8. From the Structure window, right-click the switcher component and choose **Insert Inside Switcher > Facet**.
9. In the Insert Facet dialog, name the facet `GROUP`. This will create the facet that will contain the attributes that belong to a group.
10. Drag and drop an **Iterator** as a child to the GROUP `facet` 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, `#{column.descriptors}`.
    - **Var**: A `String` that can be used to access the attributes in the descriptors, for example: `nestedCol`
12. Drag and drop a **Column** as a child to the iterator component. In the Properties window, bind **HeaderText** to the `label` property of the attributes returned by the parent iterator, for example, `#{nestedCol.label}`.
13. Drag and drop a **Dynamic Component** as a child to the second iterator.
14. In the Properties Window, set the following
    - **AttributeModel**: An EL expression that resolves to the variable created for the second iterator, for example: `#{nestedCol}`.
    - **Value**: An EL expression that resolves to each attribute name returned by the second iterator, for example:

      ```
      #{row[nestedCol.name]}
      ```
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: `#{column}`.
    - **Value**: An EL expression that resolves to each attribute name returned by the main iterator, for example:

      ```
      #{row[column.name]}
      ```

21.5 Using Validation and Conversion with Dynamic Components

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. For more information, see Section 21.1, "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. Example 21–14 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.

Example 21–14 Validation and Conversion for a Dynamic Component

```
<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="maxmum length is 10"/>
  <af:validateLongRange disabled="#{attr.name == 'Sal' ? false : true}"
                        minimum="1000"/>
</af:dynamicComponent>
```

21.6 Using Dynamic and Static Components Together

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

Example 21–15 shows how you might create the form using static `inputText` components and dynamic components that access the attributes in the Name and Address groups.

Example 21–15 Static Components Before Dynamic Components

```
<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:iterator
             value="#{DynCompBean.attributesModel.hierarchicalAttributes("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: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="dc1"
                        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

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 Example 21–16.

Example 21–16 Dynamic Components With Static Components

```
<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="dc1" 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 contains the following chapters:

22

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:

- Section 22.1, "About ADF Data Visualization Components"
- Section 22.2, "Common Functionality in Data Visualization Components"
- Section 22.3, "Providing Data for ADF Data Visualization Components"

22.1 About ADF Data Visualization Components

The ADF Data Visualization components provide significant graphical and tabular capabilities for displaying and analyzing data. 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.
- Each component offers live data preview during design. This feature is especially useful to let you see the effect of your design as it progresses without having to compile and run a page.

Data visualization components include: graphs, gauges, pivot tables and pivot filter bars, geographic maps, thematic maps, Gantt charts, timelines, hierarchy viewers, treemaps, and sunbursts.

The prefix `dvt:` occurs at the beginning of each gauge component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

22.1.1 Graph Component Use Cases and Examples

The graph component gives you the capability of producing more than 50 types of graphs, including a variety of bar graphs, pie graphs, line graphs, scatter graphs, and stock graphs. This component lets you evaluate multiple data points on multiple axes in many ways. For example, a number of graphs assist you in the comparison of results from one group against the results from another group.

The following kinds of graphs can be produced by the graph component:

- Area graph (`areaGraph`): Creates a graph in which data is represented as a filled-in area. Use area graphs to show trends over time, such as sales for the last 12 months. Area graphs require at least two groups of data along an axis. The axis is often labeled with increments of time such as months.
- Bar graph (`barGraph`): Creates a graph in which data is represented as a series of vertical bars. Use to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.
- Bar (horizontal) graph (`horizontalBarGraph`): Creates a graph that displays bars horizontally along the Y-axis. Use to provide an orientation that allows you to show trends or compare values.
- Bubble graph (`bubbleGraph`): Creates a graph in which data is represented by the location and size of round data markers (bubbles). Use 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 graph to plot salaries (X-axis), years of experience (Y-axis), and productivity (size of bubble) for your work force. Such a graph allows you to examine productivity relative to salary and experience.
- Combination graph (`comboGraph`): Creates a graph that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use to compare bars and lines, bars and areas, lines and areas, or all three.
- Funnel graph (`funnelGraph`): Creates a graph that is a visual representation of data related to steps in a process. The steps appear as vertical slices across a horizontal cylinder. As the actual value for a given step or slice approaches the quota for that slice, the slice fills. Typically a funnel graph requires actual values and target values against a stage value, which might be time. For example, use this component to watch a process (such as a sales pipeline) move towards a target across the stage of the quarters of a fiscal year.
- Line graph (`lineGraph`): Creates a graph in which data is represented as a line, as a series of data points, or as data points that are connected by a line. Line graphs require data for at least two points for each member in a group. For example, a line graph over months requires at least two months. Typically a line of a specific color is associated with each group of data such as Americas, Europe, and Asia. Use to compare items over the same time.
- Pareto graph (`paretoGraph`): Creates a graph in which data is represented by bars and a percentage line that indicates the cumulative percentage of bars. Each set of bars identifies different sources of defects, such as the cause of a traffic accident. The bars are arranged by value, from the largest number to the lowest number of incidents. A Pareto graph is always a dual-Y graph in which the first Y-axis corresponds to values that the bars represent and the second Y-axis runs from 0 to 100% and corresponds to the cumulative percentage values. Use the Pareto graph to identify and compare the sources of defects.
- Pie graph (`pieGraph`): Creates a graph in which one group of data is represented as sections of a circle causing the circle to look like a sliced pie. Use to show the relationship of parts to a whole such as how much revenue comes from each product line.
- Radar graph (`radarGraph`): Creates a graph that appears as a circular line graph. Use to show patterns that occur in cycles, such as monthly sales for the last three years.

- Scatter/polar graph (`scatterGraph`): Creates a graph in which data is represented by the location of data markers. Use to show correlation between two different kinds of data values such as sales and costs for top products. Scatter graphs are especially useful when you want to see general relationships among a number of items.
- Sparkchart (`sparkChart`): Creates a simple, condensed graph that displays trends or variations, often in the column of a table or inline with text. Sparkcharts are simple in design, with limited features and formatting options, showing as much data as possible.
- Stock graph (`stockGraph`): Creates a graph in which data shows the high, low, and closing prices of a stock. Each stock marker displays three separate values.

Figure 22–1 show an application dashboard that illustrates:

- bar graphs with and without reference lines
- pie graph

Figure 22–1   Dashboard with Bar Graph and Pie Graph

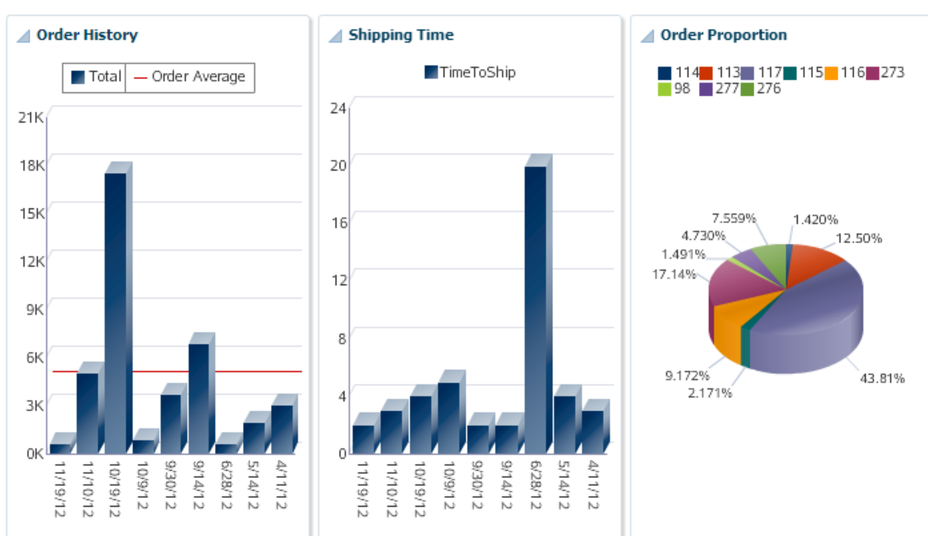


Figure 22–2 shows an application dashboard that provides examples of most graph types.

Figure 22–2 Dashboard Illustrating Multiple Graph Types

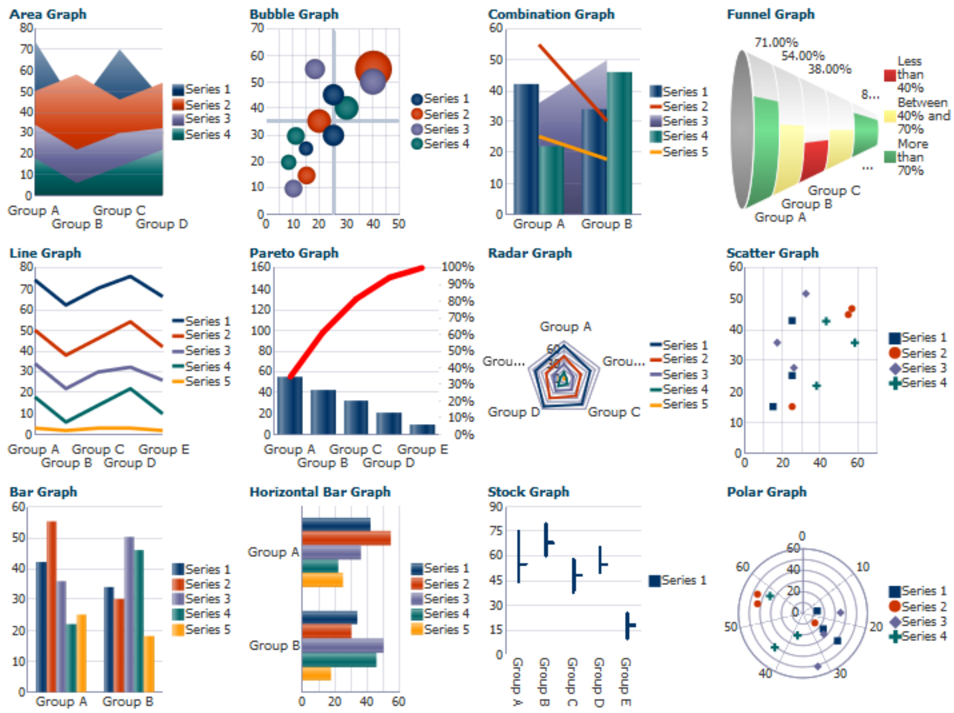


Figure 22–3 shows a line sparkchart displaying sales trends in a table column.

Figure 22–3 Sparkchart of Sales Trends

For more information including additional use cases and examples, see Chapter 23, "Using Graph Components."

22.1.2 Gauge Component Use Cases and Examples

The gauge (`gauge`) component renders graphical representations of data. Unlike the graph, a gauge focuses on a single data point and examines that point relative to minimum, maximum, and threshold indicators to identify problem areas.

One gauge component can create a single gauge or a set of gauges depending on the data provided.

The following kinds of gauges can be produced by this component:

- Dial gauge: Creates a gauge that indicates its metric value along an 180-degree arc. This type of gauge usually has an indicator in the shape of a line or an arrow that points to the value that the gauge is plotting.

- Status meter gauge: Creates a gauge that indicates the progress of a task or the level of some measurement along a horizontal rectangular bar. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle.
- Status meter gauge (vertical): Creates a gauge that indicates the progress of a task of the level of some measurement along a vertical rectangular bar.
- LED (lighted electronic display) gauge: Creates a gauge that depicts graphically a measurement, such as key performance indicator (KPI). Several styles of graphics are available for LED gauges such as arrows that indicate good (up arrow), fair (left- or right-pointing arrow), or poor (down arrow).

You can specify any number of thresholds for a gauge. However, some LED gauges (such as those with arrow or triangle indicators) support a limited number of thresholds because there are a limited number of meaningful directions for them to point. For arrow or triangle indicators, the threshold limit is three.

Figure 22–4 shows a set of dial gauges set with thresholds to display warehouse stock levels.

Figure 22–4 Dial Gauges Set with Thresholds

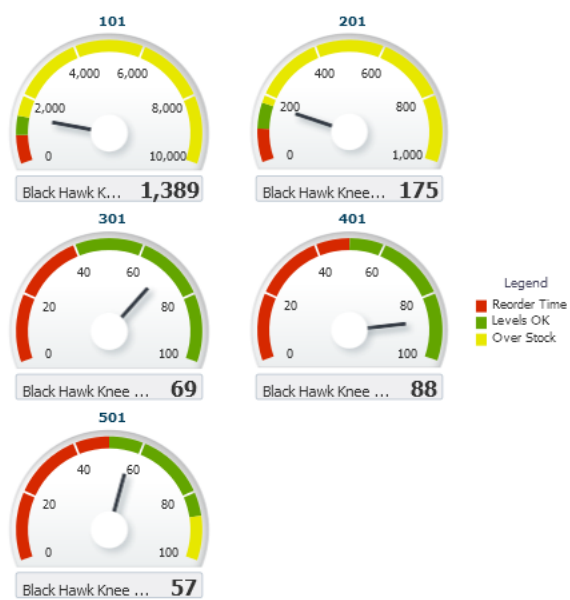
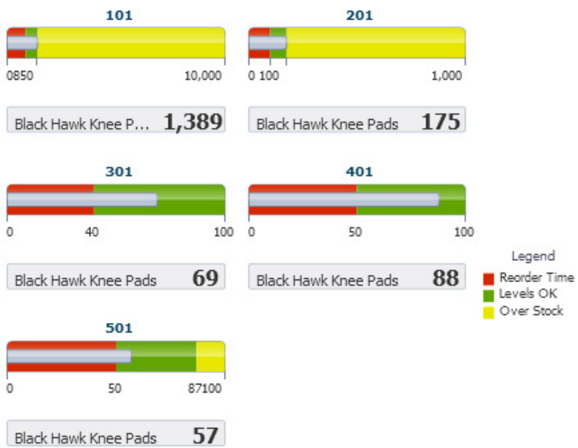


Figure 22–5 shows the same data displayed as a status meter gauge set with thresholds.

Figure 22–5 Status Meter Gauges Set with Thresholds



For more information including additional use cases and examples, see Chapter 24, "Using Gauge Components."

22.1.3 Pivot Table 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 22–6 shows an example pivot table with a pivot filter bar.

Figure 22–6 Pivot Table with Pivot Filter Bar

Filters: Direct

		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

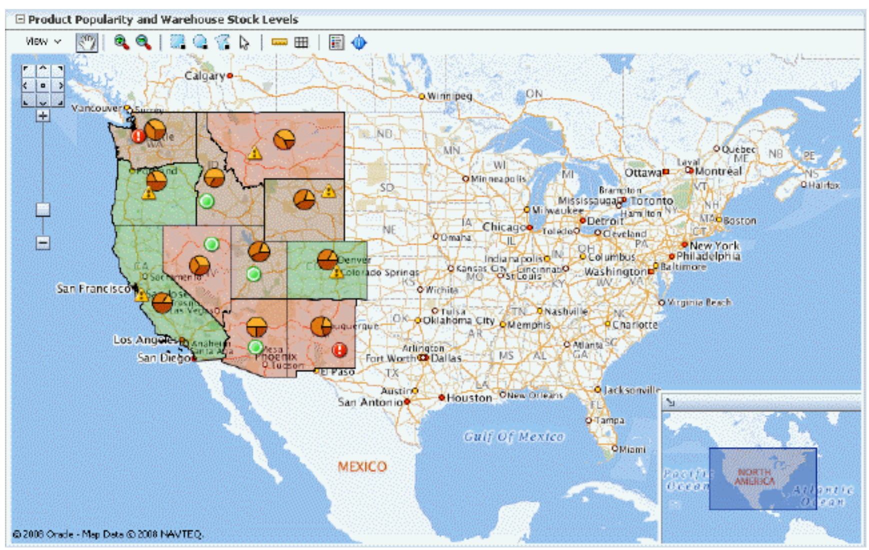
For more information including additional use cases and examples, see Chapter 25, "Using Pivot Table Components."

22.1.4 Geographic Map 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 22–7 shows a geographic map with color theme, pie graph theme, and point theme.

Figure 22–7 Geographic Map with Color Theme, Pie Graph Theme, and Point Theme



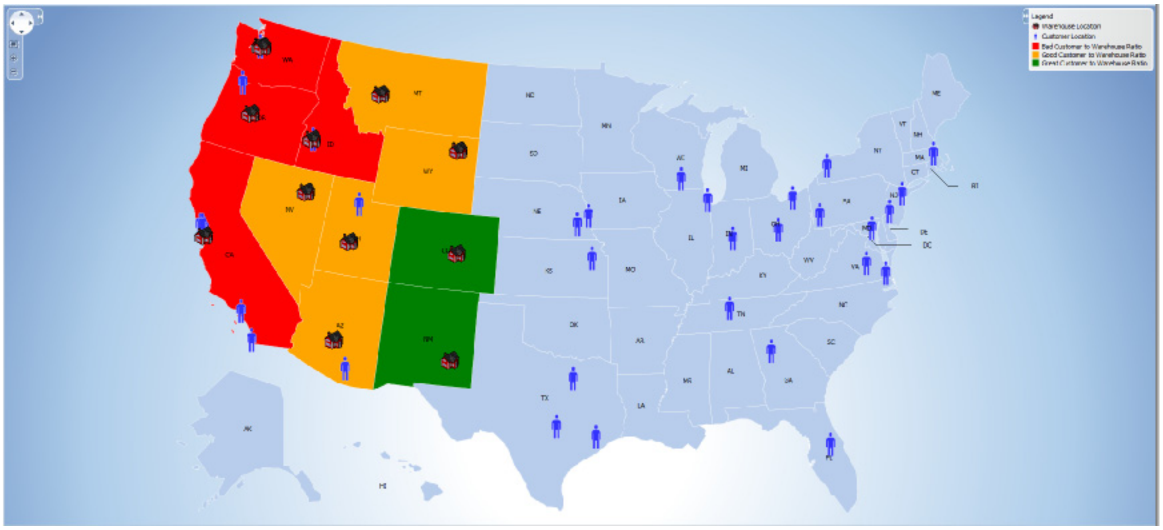
For more information including additional use cases and examples, see Chapter 28, "Using Map Components."

22.1.5 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 22–8. The example illustrates thematic map default features including a data bound legend.

Figure 22–8 Thematic Map Displaying Warehouse to Customer Ratios



For more information including additional use cases and examples, see Chapter 28, "Using Map Components."

22.1.6 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.

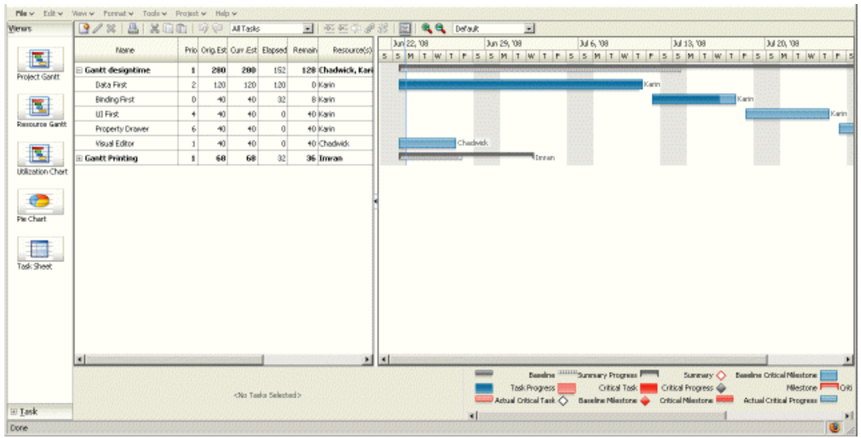
A Gantt chart consists of two ADF Faces tree tables combined with a splitter. The left-hand table contains a list of tasks or resources while the right-hand table consists of a single column in which progress is graphed over time.

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 22–9 shows a project Gantt view of staff resources and schedules.

Figure 22–9 Project Gantt Chart



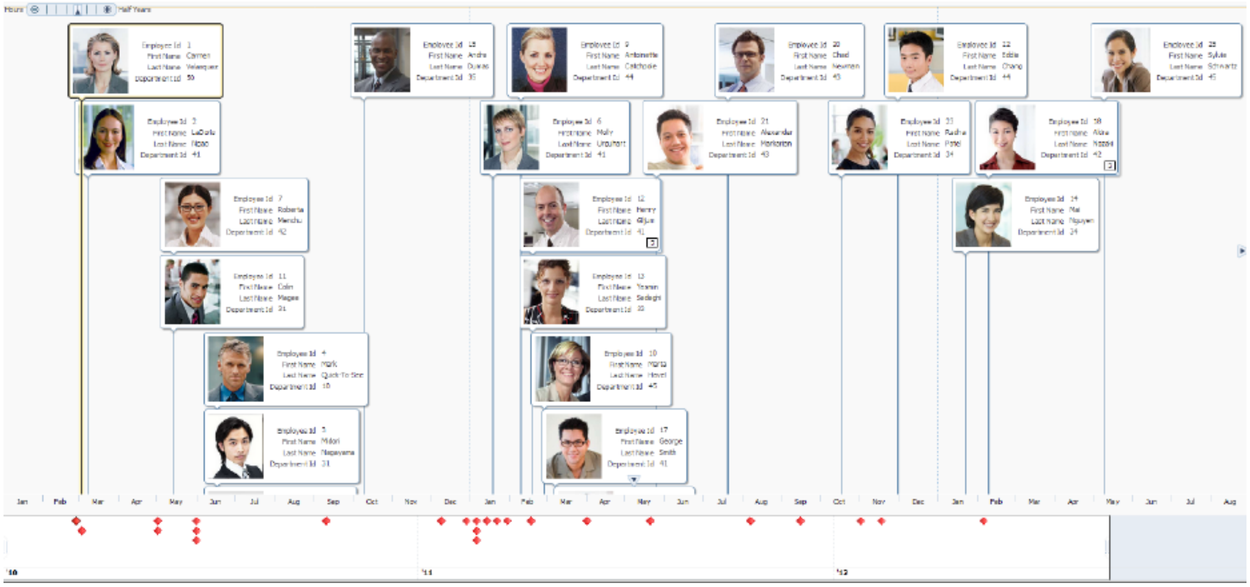
For more information including additional use cases and examples, see Chapter 26, "Using Gantt Chart Components."

22.1.7 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.

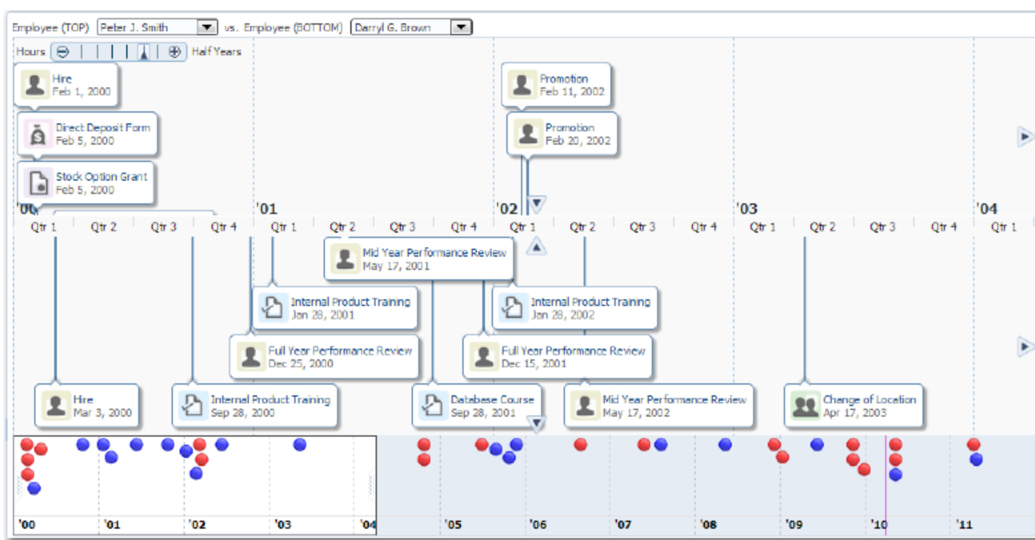
For example, the timeline in Figure 22–10 is configured to display the chronological order of the hire dates of employees. In this example, timeline items representing each event display information about the employee using an image and text with labels. The overview window defines the time range for 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 22–10 Timeline of Employee Hire Dates



A dual timeline can be used for comparison of up to two series of events. Figure 22–11 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 red colored line in the overview time axis indicates the current date.

Figure 22–11 Dual Timeline Comparing Employee Change Events

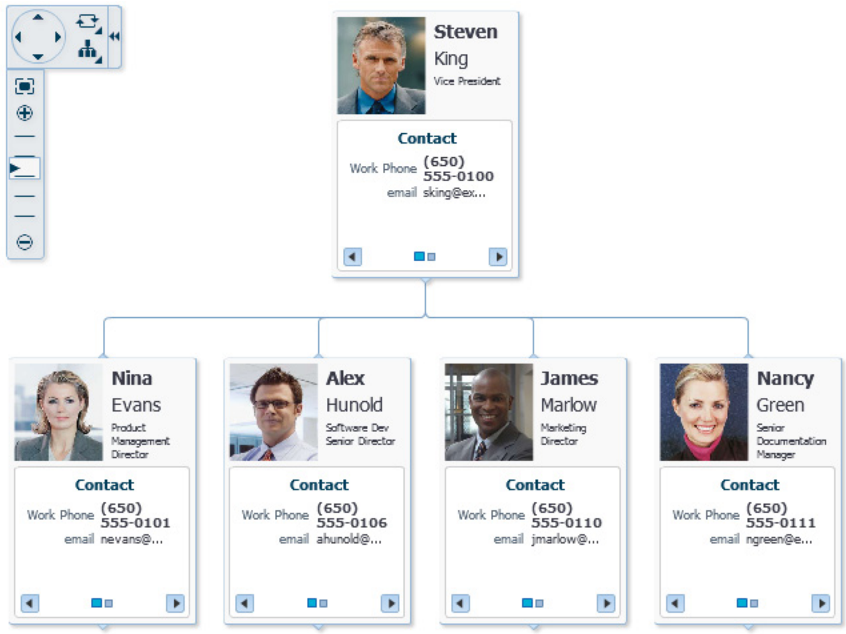


For more information including additional use cases and examples, see Chapter 27, "Using Timeline Components."

22.1.8 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, as shown in Figure 22–12.

Figure 22–12 Hierarchy Viewer as Organizational Chart



For more information including additional use cases and examples, see Chapter 29, "Using Hierarchy Viewer Components."

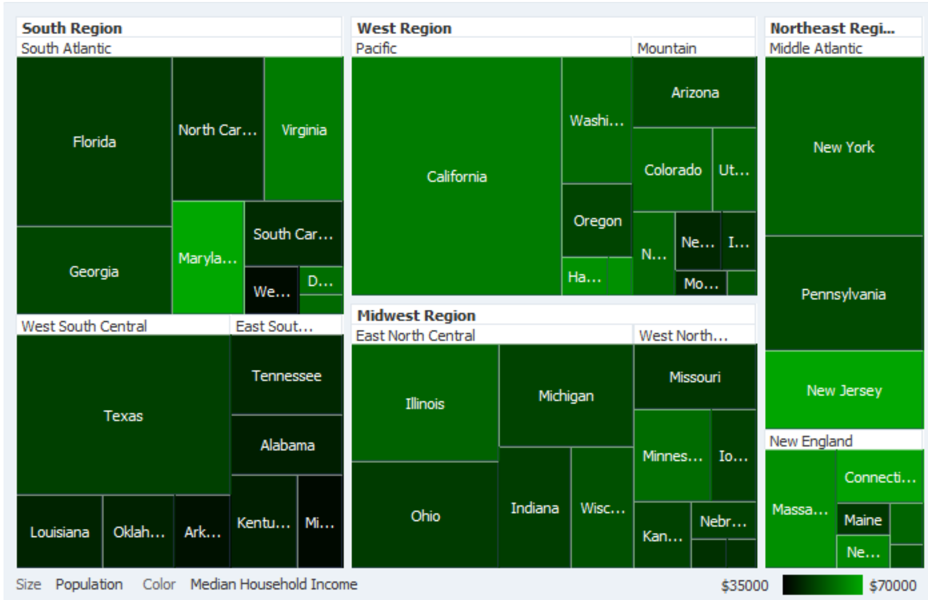
22.1.9 Treemap and Sunburst Components Use Cases and Examples

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.

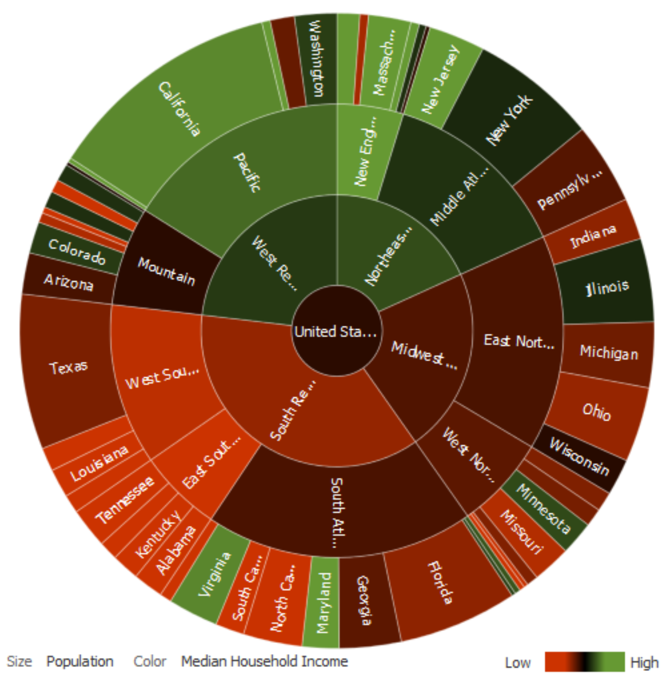
Figure 22–13 shows a treemap displaying United States census data grouped by regions, with the color attribute used to indicate median income levels. States with larger populations display in larger-sized nodes than states with smaller populations.

Figure 22–13 Treemap Displaying United States Census Data by Region



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 22–14 shows the same census data displayed in a sunburst.

Figure 22–14 Sunburst Displaying United States Census Data by Region



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.

For additional information about treemaps and how to use them in your application, see Chapter 30, "Using Treemap and Sunburst Components."

22.1.10 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: You can make your data visualization components accessible. For more information, see Chapter 33, "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 Appendix D, "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 Chapter 31, "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*.

22.2 Common Functionality in Data Visualization Components

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.

22.2.1 Content Delivery

Data visualization components including gauge, graph, 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 lifecycle. 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 Section 12.2.2, "Content Delivery."

22.2.2 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 Section 8.1, "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*.

22.2.3 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. For graph and gauge, support is limited to a subset of the graph and gauge types.

Table 22–1 DVT Components Supporting Active Data

DVT Component	Link to Component Detail
gauge	Section 24.6.6, "How to Configure Gauges to Display Active Data"
geographic map	Section 28.2.4, "What You May Need to Know About Active Data Support for Map Point Themes"
graph	Section 23.6.1, "How to Configure Databound Graph Components to Display Active Data"
pivot table and pivot filter bar	Section 25.1.2.11, "Active Data Support (ADS)"
sunburst	Section 30.1.2.16, "Active Data Support (ADS)"
treemap	Section 30.1.2.16, "Active Data Support (ADS)"

For additional information about using the Active Data Service, see Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

22.2.4 Context Menus for Graphs, Gauges, Treemaps, and Sunbursts

Context menus can be defined for `graph`, `gauge`, `treemap`, and `sunburst` components using these context menu facets:

- `bodyContextMenu`: Specifies a context menu that is displayed on elements in the component that are not selectable.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the component.
- `multiSelectContextMenu`: Specifies a content menu that is displayed when multiple elements are selected in the component.

Each facet supports a single child component. For all of these facets to work, selection must be enabled and supported for the specific graph or gauge type. Context Menus are currently only supported in Flash and HTML5.

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 it 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`.

22.2.5 Screen Reader Support

All data visualization components provide attributes that support descriptive text about the component. Setting the appropriate attribute for the component is required for supporting the screen reader mode. Set the following attributes for data visualization components:

- `shortDesc` attribute: Use for all graph and gauge components.
- `summary` attribute: Use for pivot table, Gantt chart, geographic and thematic map, hierarchy viewer, sunburst, timeline, and treemap components.

22.2.6 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 `pieGraph` component by setting the `text` attribute on its child `graphTitle` component, 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 Section 32.2, "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.

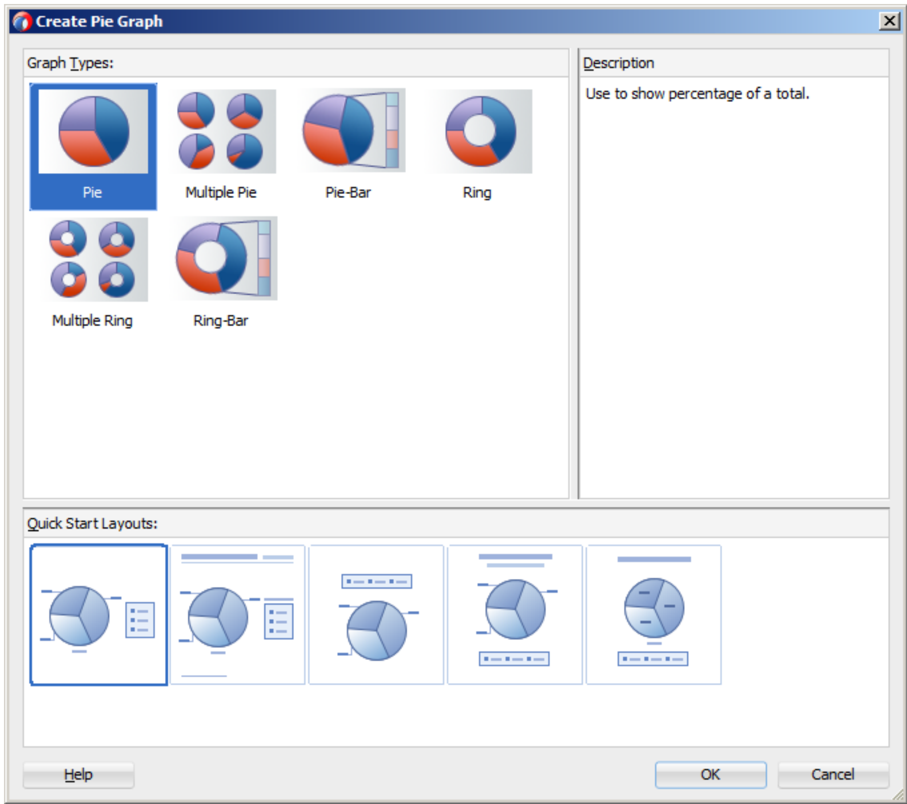
22.3 Providing Data for ADF Data Visualization Components

In JDeveloper you can add any 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.

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, when you are designing your page using simple UI-first development, you use the Components window to add a graph to a JSF page. When you drag and drop a graph component onto the page, the Create Graph dialog displays available categories of graph types, with descriptions, to provide visual assistance when creating graphs. You can also specify a quick start layout of the graph's title and legend. Figure 22–15 shows the Create Pie Graph that displays when creating a pie graph from the Components window.

Figure 22–15 Create Pie Graph Dialog Dragged from Components Window



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:

- Chapter 23, "Using Graph Components"
- Chapter 24, "Using Gauge Components"

> **Note:** Graphs and gauges have a `tabularData` method that lets you provide CSV (Comma Separated Value) data from a method stored in a managed bean.

- Chapter 25, "Using Pivot Table Components"
- Chapter 26, "Using Gantt Chart Components"
- Chapter 27, "Using Timeline Components"
- Chapter 28, "Using Map Components"
- Chapter 29, "Using Hierarchy Viewer Components"

- Chapter 30, "Using Treemap and Sunburst Components"

You can also create and data bind a graph by dragging a data control from the Data Controls Panel. A Component Gallery displays available graph categories, types, and descriptions to provide visual assistance when designing graphs and defining a quick layout. Figure 22–16 shows the Component Gallery that displays when creating a line graph from a data control.

Figure 22–16 Component Gallery for Line Graphs from Data Controls Panel

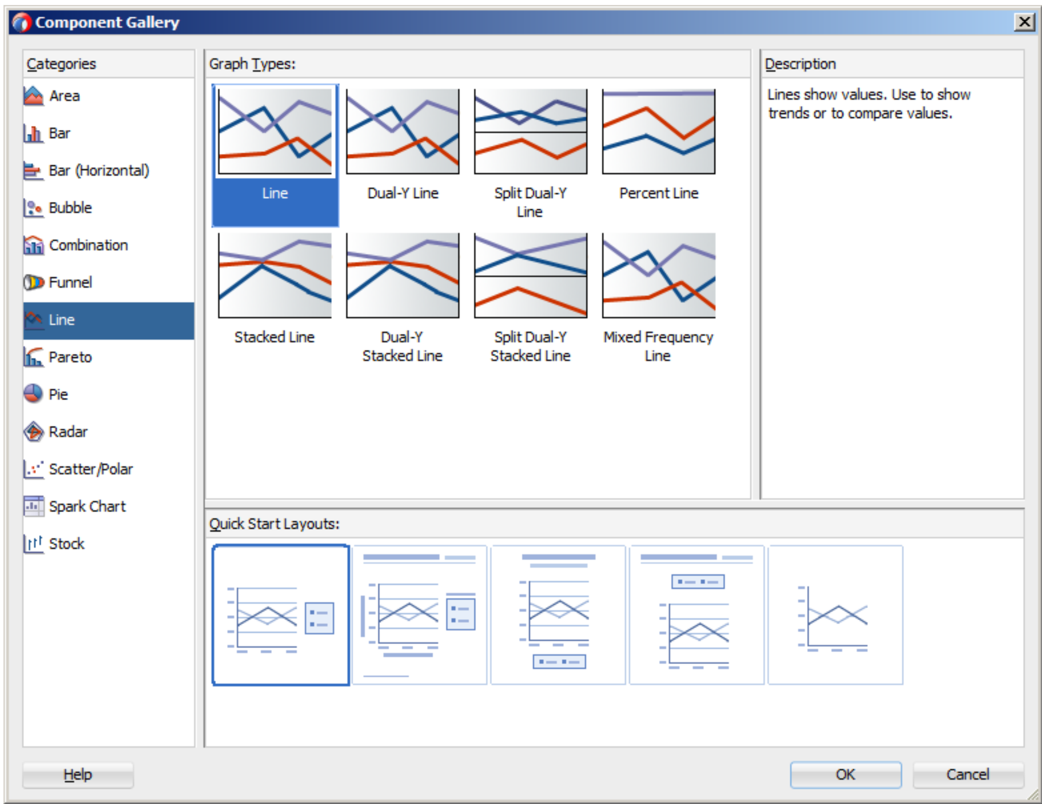
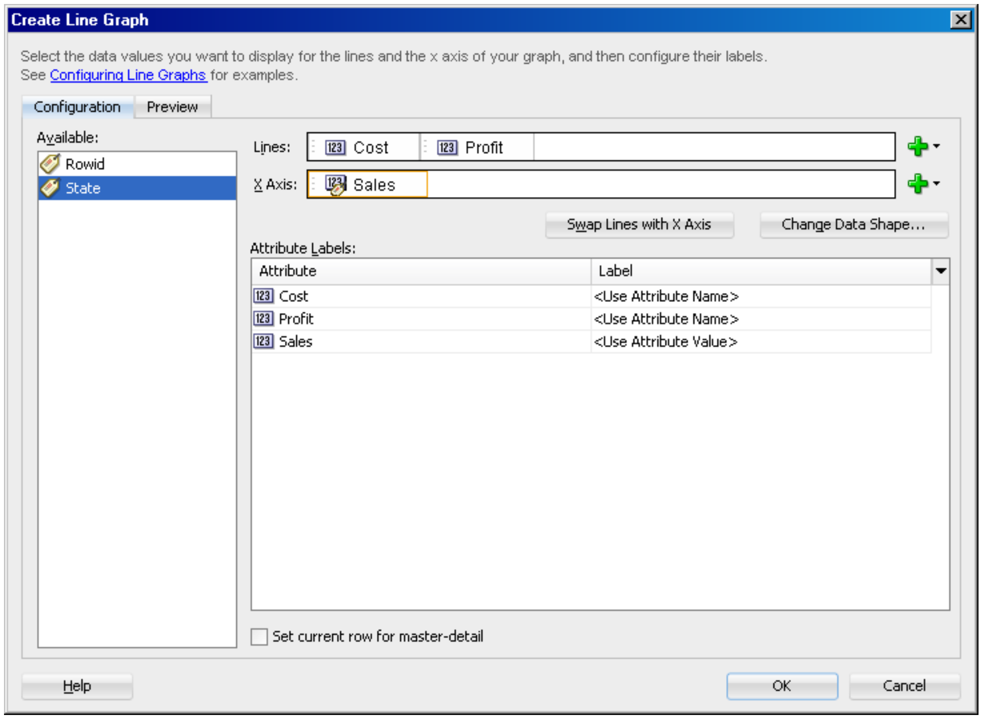


Figure 22–17 shows the Create Line Graph dialog you use to bind the data collection attributes to the graph component.

Figure 22–17 Create Line Graph Data Binding 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 Graphs" 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 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 Viewers" 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*

23

Using Graph Components

This chapter describes how to use ADF Data Visualization `areaGraph`, `barGraph`, `horizontalBarGraph`, `bubbleGraph`, `comboGraph`, `funnelGraph`, `lineGraph`, `paretoGraph`, `pieGraph`, `radarGraph`. `scatterGraph`, `sparkChart`, and `stockGraph` components to display data in graphs 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 also use data controls to create graphs. For more information, see the "Creating Databound Graphs" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

23.1 About the Graph Component

The graph component gives you the capability of producing more than 50 types of graphs, including a variety of area, bar, bubble, combination, funnel, line, Pareto, pie, radar, scatter, sparkchart, and stock graphs. This component lets you evaluate multiple data points on multiple axes in many ways. For example, a number of graphs assist you in the comparison of results from one group with the results from another group.

A graph displays *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 graph and the columns in the grid appear as groups in the graph.

For most graphs, a series appears as a set of markers that are the same color. Typically, the graph legend shows the identification and associated color of each series. For example, in a bar graph, the yellow bars might represent the sales of shoes and the green bars might represent the sales of boots.

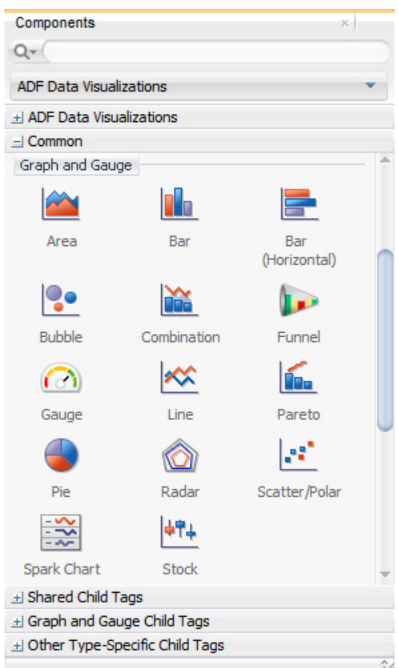
Groups appear differently in different graph types. For example, in a clustered bar graph, each cluster is a group. In a stacked bar graph, each stack is a group. In a multiple pie graph, each pie is a group. A group might represent time periods, such as years. A group might also represent geographical locations such as regions.

Depending on the data requirements for a graph type, a single group might require multiple data values. For example, a scatter graph 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.

23.1.1 Graph Component Use Cases and Examples

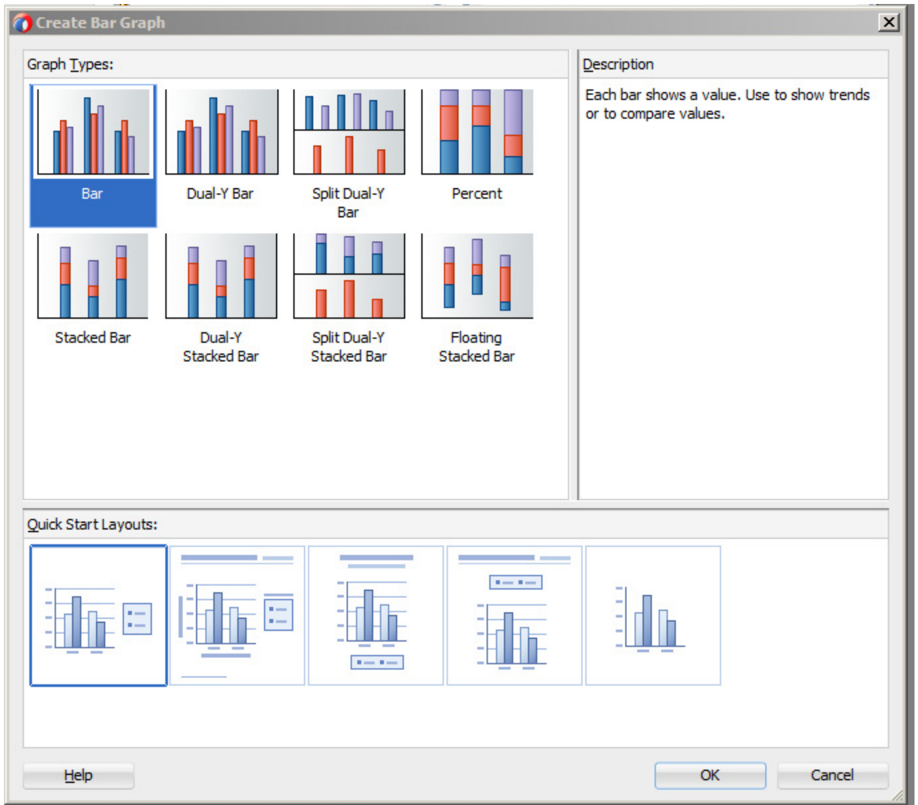
Graph components include 13 types of graphs with one or more variations for a total of over 50 different graphs you can use to display data. JDeveloper provides a Components window that displays available graph categories. Figure 23–1 shows the Components window for graphs.

Figure 23–1 Components Window for Graphs



When you select a graph category in the Components window, JDeveloper displays a dialog with descriptions about the available graph types to provide visual assistance when you are creating graphs. Figure 23–2 shows the different bar graph types and layouts available when you select the Bar graph category in the Components window.

Figure 23–2 Bar Graph Types in Create Bar Graph Dialog



Graph types include:

- Area: Represents data as a filled-in area. Use area graphs to show trends over time, such as sales for the last 12 months. Area graphs require at least two groups of data along an axis. The axis is often labeled with increments of time such as months.

  Area graphs represent these kinds of data values:

  – Absolute: Each area marker connects a series of two or more data values. This type of graph has the following variations: Absolute area graph with a single y-axis and absolute area graph with a split dual-Y axis.

    In a split dual-Y graph, the plot area is split into two sections, so that sets of data assigned to the different Y-axes appear in different parts of the plot area.

  – 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. This kind of graph has the following variations: Stacked area graph with a single y-axis and stacked area graph with a split dual y-axis.

  – Percentage: Area markers show the percentage of the cumulative total of all sets of data.

  Figure 23–3 shows variations of the area graph type as displayed in the Create Area Graph dialog with the default graph selected.

Figure 23–3 Area Graph Type Variations

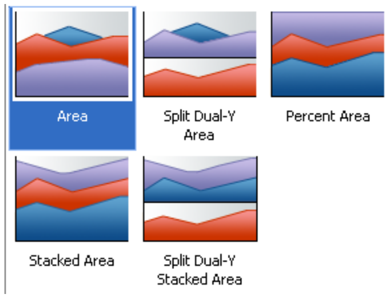
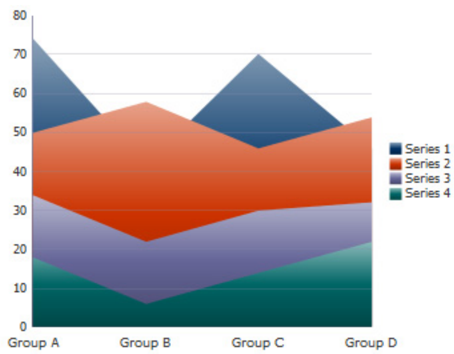


Figure 23–4 shows an example area graph.

Figure 23–4 Area Graph Example



- Bar: Represents data as a series of vertical bars. Use bar graphs to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.

  Bar graphs 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 graph includes the following variations: vertical clustered bar graphs and horizontal clustered bar graphs. All variations of clustered bar graphs can be arranged as single y-axis, dual y-axis, and split dual y-axis graphs.

  – Stacked: Bars for each set of data are appended to previous sets of data. The size of the stack represents a cumulative data total. This kind of graph includes the following variations: Vertical stacked bar graphs and horizontal stacked bar graphs. All variations of stacked bar graphs can be arranged as single y-axis, dual y-axis, and split dual y-axis graphs.

  – Percentage: Bars are stacked and show the percentage of a given set of data relative to the cumulative total of all sets of data. Percentage bar graphs are arranged only with a single y-axis.

  Figure 23–5 shows variations of the bar graph type as displayed in the Create Bar Graph dialog with the default graph selected.

Figure 23–5 Bar Graph Type Variations

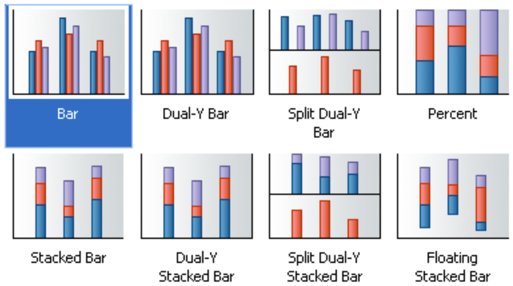
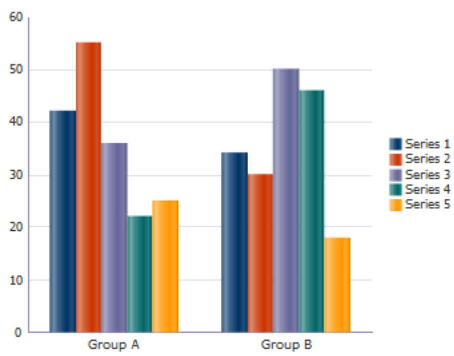


Figure 23–6 shows an example bar graph.

Figure 23–6 Bar Graph Example



- Horizontal bar: Displays bars horizontally along the y-axis. Use horizontal bar graphs to provide an orientation that allows you to show trends or compare values.

  Figure 23–7 shows variations of the horizontal bar graph type as displayed in the Create Horizontal Bar Graph dialog with the default graph selected.

Figure 23–7 Horizontal Bar Graph Type Variations

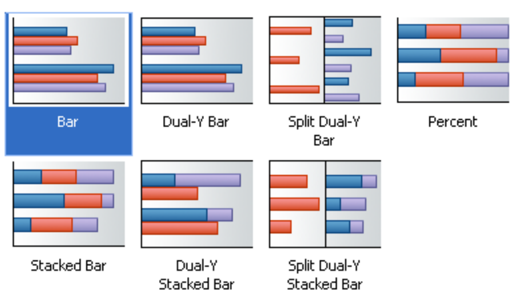
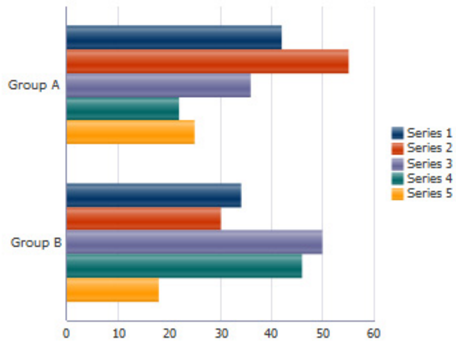


Figure 23–8 shows an example horizontal bar graph.

Figure 23–8 Horizontal Bar Graph Example



- Bubble: Represents data by the location and size of round data markers (bubbles). Use bubble graphs 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 graph to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a graph allows you to examine productivity relative to salary and experience.

  Figure 23–9 shows variations of the bubble graph type as displayed in the Create Bubble Graph dialog with the default graph selected.

Figure 23–9 Bubble Graph Type Variations

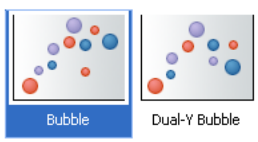
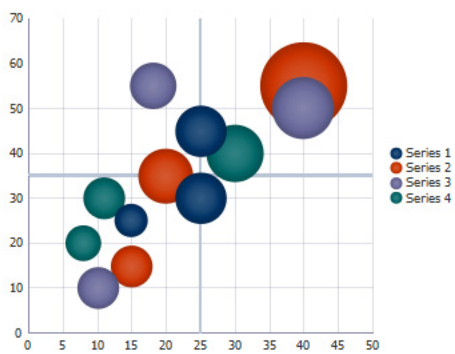


  Figure 23–10 shows an example bubble graph.

Figure 23–10 Bubble Graph Example



- Combination: Graph that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use combination graphs to compare bars and lines, bars and areas, lines and areas, or all three combinations.

  Figure 23–11 shows variations of the combination graph type as displayed in the Create Combination Graph dialog with the default graph selected.

Figure 23–11 Combination Graph Type Variations

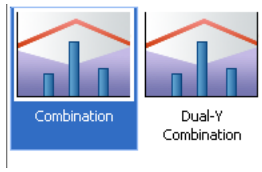
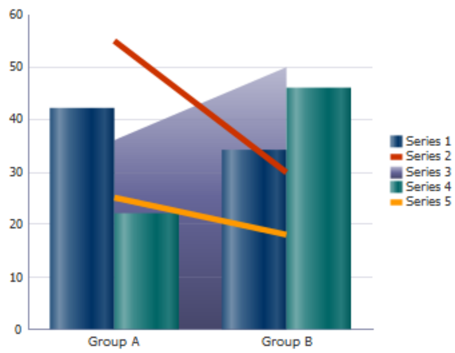


Figure 23–12 shows an example combination graph.

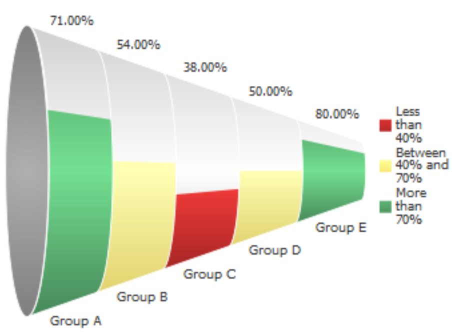
Figure 23–12 Combination Graph Example



- Funnel: Visually represents data related to steps in a process. The steps appear as vertical slices across a horizontal cone-shaped section. As the actual value for a given step or slice approaches the quota for that slice, the slice fills. Typically, a funnel graph requires actual values and target values against a stage value, which might be time. For example, use the funnel graph to watch a process where the different sections of the funnel represent different stages in the sales cycle. There are no variations of the funnel graph.

  Figure 23–13 shows an example funnel graph.

Figure 23–13 Funnel Graph Example



- Line: Represents data as a line, as a series of data points, or as data points that are connected by a line. Line graphs require data for at least two points for each member in a group. For example, a line graph 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 graphs to compare items over the same time.

  Line graphs represent these kinds of data values:

– Absolute: Each line segment connects two data points. This kind of graph can have its axes arranged as single y-axis, dual y-axis, and split dual y-axis.

– Stacked: Lines for each set of data are appended to previous sets of data. The size of the stack represents a cumulative data total. This kind of graph can have its axes arranged as single y-axis, dual y-axis, and split dual y-axis.

– Percentage: Lines are stacked and each line shows the percentage of a given set of data relative to the cumulative total of all sets of data. Percentage line graphs are arranged only with a single y-axis.

Figure 23–14 shows variations of the line graph type as displayed in the Create Line Graph dialog with the default graph selected.

Figure 23–14 Line Graph Type Variations

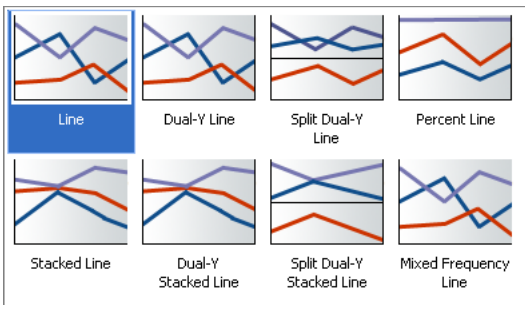
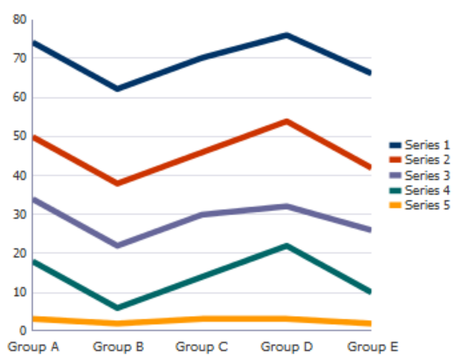


Figure 23–15 shows an example line graph.

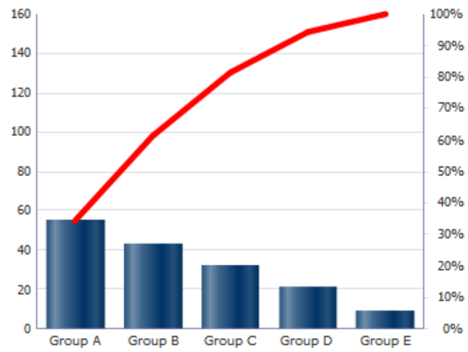
Figure 23–15 Line Graph Example



- Pareto: Represents data by bars and a percentage line that indicates the cumulative percentage of bars. Each set of bars identifies different sources of defects, such as the cause of a traffic accident. The bars are arranged by value, from the largest number to the lowest number of incidents. A Pareto graph is always a dual-Y graph in which the first y-axis corresponds to values that the bars represent and the second y-axis runs from 0% to 100% and corresponds to the cumulative percentage values. Use Pareto graphs to identify and compare the sources of defects. The Pareto graph has no variations.

  Figure 23–16 shows an example Pareto graph.

Figure 23–16 Pareto Graph Example



- Pie/Ring: Represents one group of data as sections of a circle causing the circle to look like a sliced pie. Use pie graphs to show the relationship of parts to a whole such as how much revenue comes from each product line. Pie graphs can display as a pie or ring, where the center of each circle has a hole in which the total pie value is displayed.

  Figure 23–17 shows variations of the pie graph type as displayed in the Create Pie Graph dialog with the default graph selected.

Figure 23–17 Pie Graph Type Variations

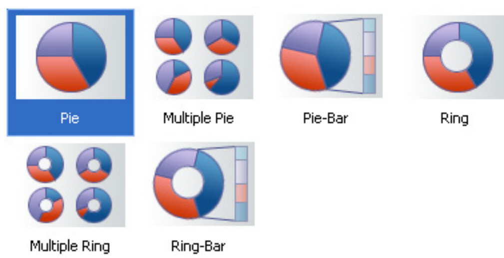
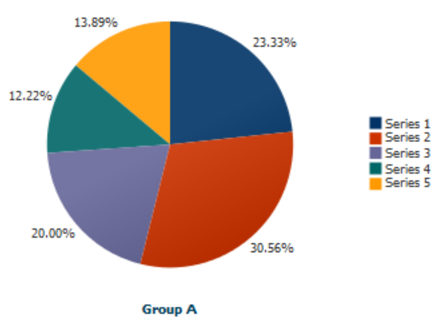


  Figure 23–18 shows an example pie graph.

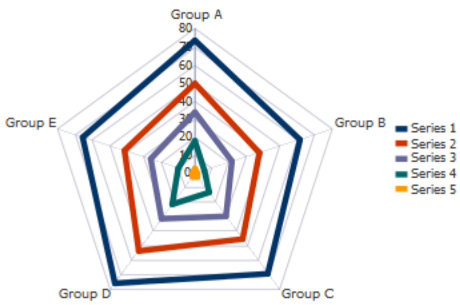
Figure 23–18 Pie Graph Example



- Radar: A graph that appears as a circular line graph. Use radar graphs to show patterns that occur in cycles, such as monthly sales for the last three years. The radar graph has no variations.

  Figure 23–19 shows an example radar graph.

Figure 23–19 Radar Graph Example



- Scatter/Polar: Represents data by the location of data markers. Use scatter graphs to show correlation between two different kinds of data values such as sales and costs for top products. Use scatter graphs in particular to see general relationships among a number of items. A scatter graph can display data in a directional manner as a polar graph.

  Figure 23–20 shows variations of the scatter graph type as displayed in the Create Scatter Graph dialog with the default graph selected.

Figure 23–20 Scatter Graph Type Variations

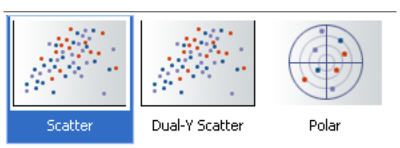
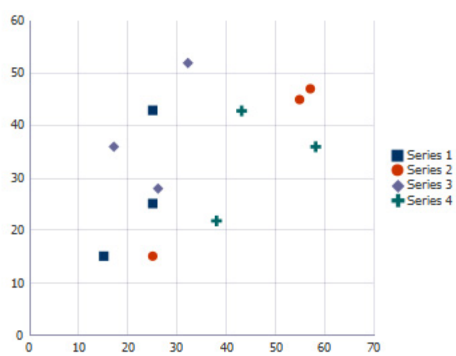


  Figure 23–21 shows an example scatter graph.

Figure 23–21 Scatter Graph Example



- Sparkchart: A simple, condensed graph that displays trends or variations in a single data value, typically stamped in the column of a table or in line with related text. Sparkcharts have basic conditional formatting. Since sparkcharts contain no labels, the adjacent columns of a table or surrounding text provide context for sparkchart content

  Figure 23–22 shows variations of the sparkchart graph type as displayed in the Create Sparkchart Graph dialog with the default graph selected.

Figure 23–22 Sparkchart Graph Type Variations

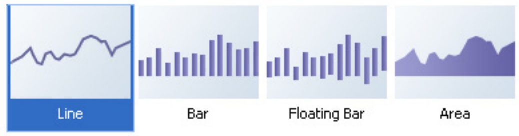


Figure 23–23 shows an example sparkchart.

Figure 23–23 Bar Sparkchart Example

- Stock: Shows data as the high, low, and closing prices of a stock. Each stock marker displays two to four separate values (not counting the optional volume marker) depending on the specific type of stock graph chosen. Stock graphs display stock prices and, optionally, the volume of trading for one or more stocks in a graph. When any stock or candle stock graph includes the volume of trading, the volume appears as bars in the lower part of the graph.

  Candle stock graphs display stock prices and, optionally, the volume of trading for only a single stock. When a candle stock graph includes the volume of trading, the volume appears as bars in the lower part of the graph.

  Candle stock graphs also show the lesser of the open and close values at the bottom of the candle. The greater value appears at the top of the candle. If the closing value is greater than the opening value, then the candle is green. If the opening value is higher than the closing value, then the candle is red.

  Figure 23–24 shows variations of the stock graph type as displayed in the Create Stock Graph dialog with the default graph selected.

Figure 23–24 Stock Graph Type Variations

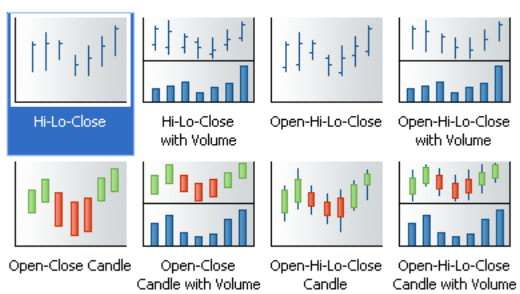
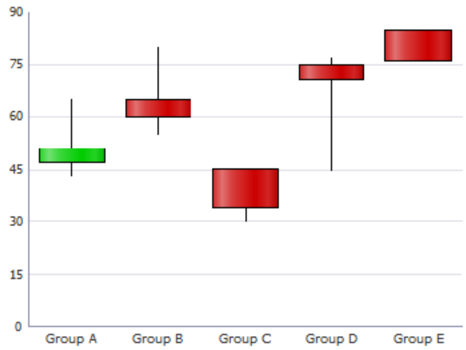


Figure 23–25 shows an example candle stock graph.

Figure 23–25 Candle Stock Chart Example



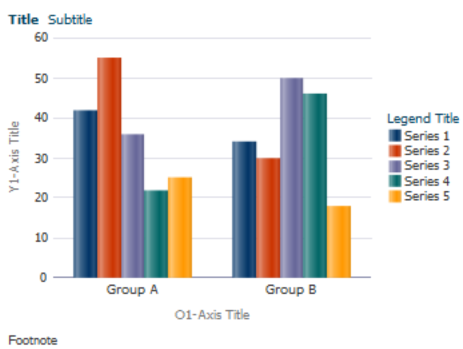
23.1.2 End User and Presentation Features

Graph end user and configurable presentation features include a rich variety of options.

23.1.2.1 Graph Layout

The optional graph title, subtitle, footnote, legend, and axis title components can be customized for placement and appearance. The plot area, present for all graphs, can be customized for appearance. Figure 23–26 shows the default display of those graph components for a bar graph.

Figure 23–26 Graph Layout Components



23.1.2.2 Sizing

Graphs are displayed in a default size of 400 X 300 pixels. You can customize the size of a graph or specify dynamic resizing to fit an area across different browser window sizes. When graphs are displayed in a horizontally or vertically restricted area, for example in a web page sidebar, the graph is displayed in a fully featured, although simplified display.

23.1.2.3 Image Formats

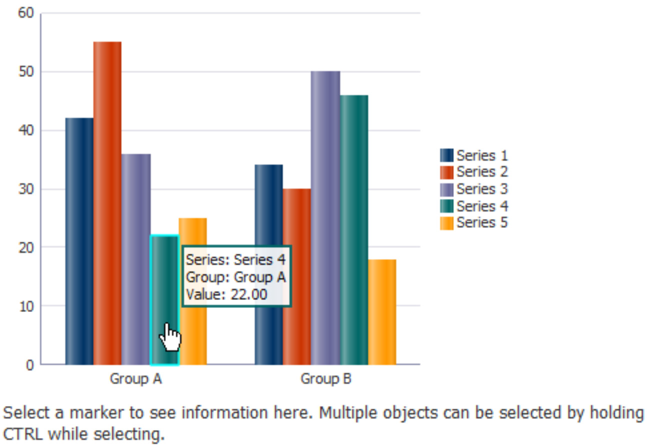
Graphs support the following image formats: HTML5, Flash, and PNG. By default, new applications default to HTML5, but you can change the default image format. You can also disable Flash across your application or customize the Flash Player's behavior on client platforms.

23.1.2.4 Data Marker Selection

Graphs can be enabled for single or multiple selection of data markers such as bubbles in a bubble graph or shapes in a scatter graph. Enabling selection is required for context menus and for responding programmatically to user clicks on the data markers.

Figure 23–27 shows a bar graph enabled for selection. Each data marker is highlighted as the user moves over it to provide a visual clue that the marker is selectable.

Figure 23–27 Bar Graph With Selection Support Enabled



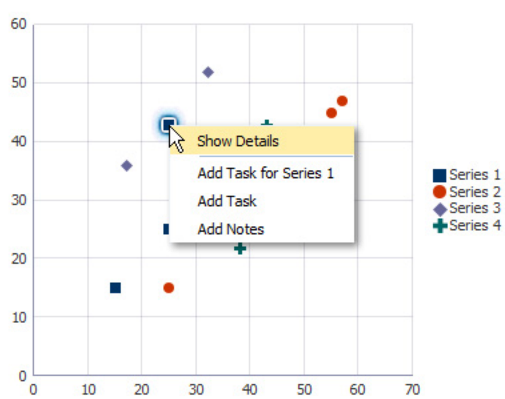
23.1.2.5 Context Menus

Graphs support right-click context menus using facets for any of three types:

- Context menus displayed on any non selectable area within the component, for example, the plot area
- Context menus displayed on any selectable element, for example, the marker in a scatter graph
- Context menus displayed on multiple selectable elements

Figure 23–28 shows a context menu displayed on a marker selected in a scatter graph.

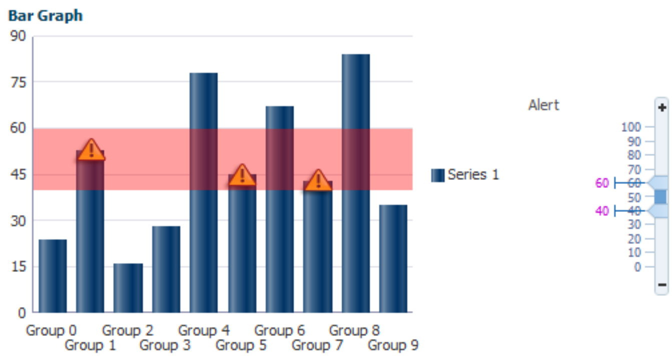
Figure 23–28 Selected Marker Context Menu in Scatter Graph



23.1.2.6 Reference Areas/Line and Alerts

Graphs can be configured to associate a data reference area or line with a graph series or axis. Separately, graphs can be configured to define an additional data point that needs to be highlighted with a separate symbol, such as for an error or warning. Figure 23–29 shows a bar graph with a warning alert icon for each bar inside the alert range of the graph. The figure also illustrates an ADF input range slider that can be associated with the graph to change the reference area.

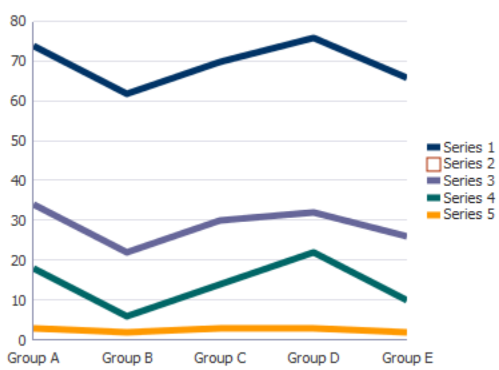
Figure 23–29 Alerts in Bar Graph



23.1.2.7 Hide and Show Series

Graphs can be configured to hide and show one or more series displayed in the graph data. This is useful for comparison and analysis, particularly when multiple series are displayed. Figure 23–30 shows hide and show in a line graph. The default icon for the hidden series is an empty box.

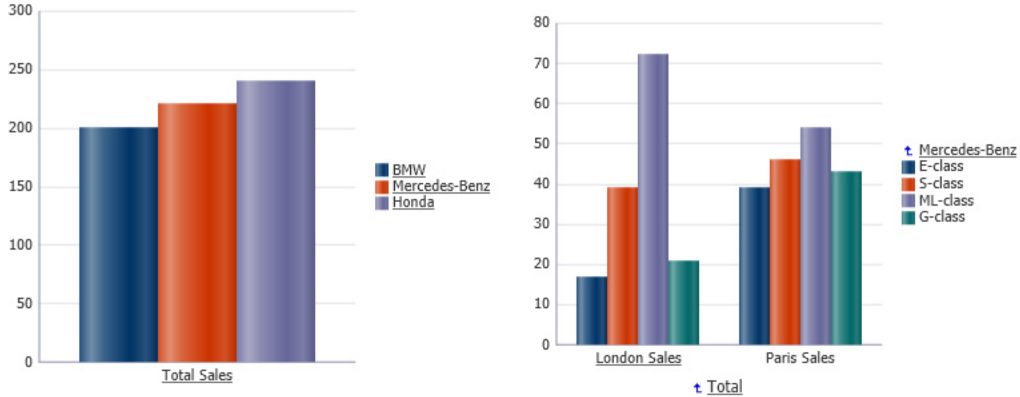
Figure 23–30 Hide and Show in Line Graph



23.1.2.8 Drilling

The series and groups in a databound graph can be configured for drilling up and down from an aggregated total to a detail view of the data. Figure 23–31 shows a bar graph displaying total sales for all cars, and a detail view of London and Paris sales for the four types of Mercedes-Benz. Active drill icons are displayed in the drilled view.

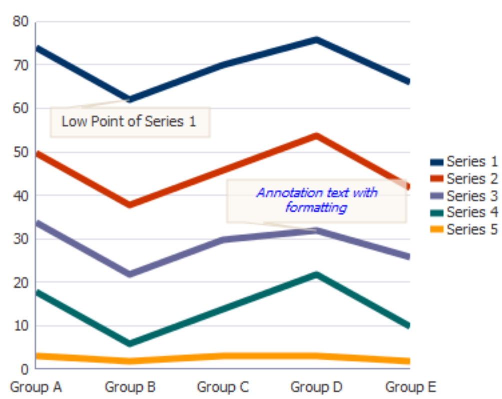
Figure 23–31 Drilling in Bar Graph



23.1.2.9 Annotations

Annotations can be used to call out significant values in the graph data. Figure 23–32 shows sample annotations in a line graph.

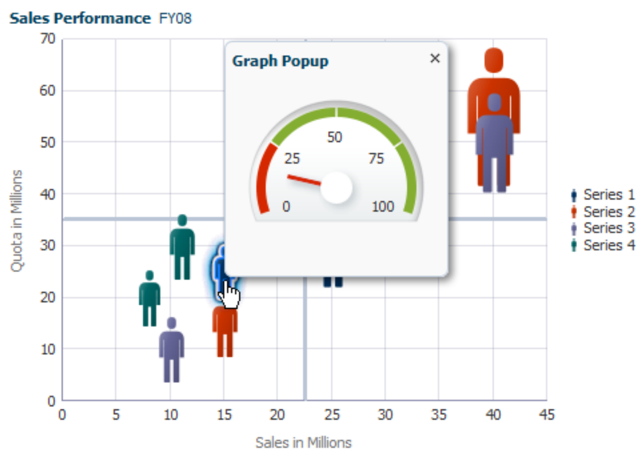
Figure 23–32 Annotations in Line Graph



23.1.2.10 Popup Support

Graph components can be configured to display popup dialogs, windows, and menus that provide information or request input from end users. Figure 23–33 shows a popup with a gauge component in a graph.

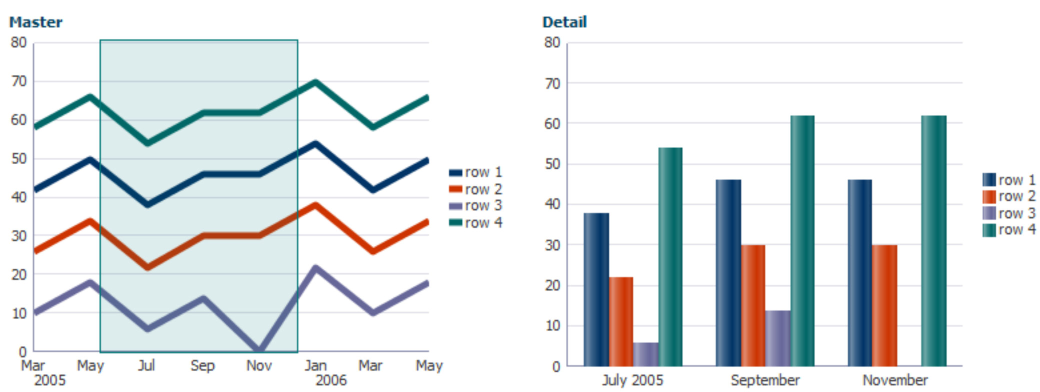
Figure 23–33 Popup in a Scatter Graph



23.1.2.11 Time Selector

Graphs that include a time axis can be configured to include a time selector which allows the user to select a time range on the time axis. Figure 23–34 shows a user-enabled time selector to display the master-detail data in graphs. When the user moves the time selector on the line graph, the bar graph changes to display the data for the selected time period.

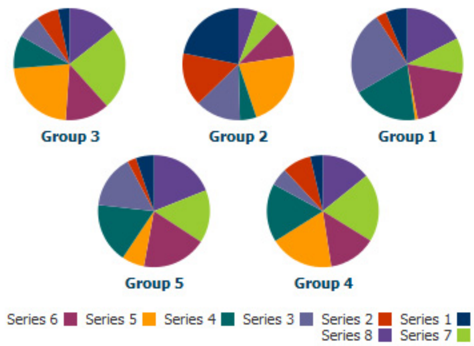
Figure 23–34 Time Selector in Line Graph



23.1.2.12 Bi-directional Support

All image formats for graphs support bi-directional locales. Figure 23–35 shows bi-directional support in multiple pie graphs.

Figure 23–35 Bi-directional Support in Multiple Pie Graphs



23.1.2.13 Drag and Drop

Graphs can be configured to support these drag and drop operations:

- Drag and drop between graphs
- Drag and drop from a graph into another ADF component
- Drag a scatter/bubble marker within the plot area of a graph
- Drag a scatter/bubble marker to another component
- Drag and drop multiple markers

23.1.2.14 Screen Reader Support

To support visually impaired users who read web pages with a screen reader, graphs are automatically replaced with pivot tables when screen reader mode is enabled for the application. Screen readers can more easily navigate and read the data in a pivot table than in a graph. For information about enabling screen reader mode, see Section 33.2, "Configuring Accessibility Support In ADF Faces." For information about ADF pivot tables, see Section 25.1, "About the Pivot Table Component."

23.1.3 Additional Functionality for Graph Components

You may find it helpful to understand other ADF Faces features before you implement your graph component. Additionally, once you have added a graph 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 graph components can use:

- Partial page rendering: You may want a graph to refresh to show new data based on an action taken on another component on the page. For more information, see Chapter 8, "Rerendering Partial Page Content."
- Personalization: When enabled, users can change the way the graph 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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: You can make your graph components accessible. For more information, see Chapter 33, "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 Appendix D, "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 graphs based on how your ADF Business Components are configured. For more information, see the "Creating Databound Graphs" section 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 Section 22.2, "Common Functionality in Data Visualization Components."

23.2 Using the Graph Component

Data requirements for graphs differ with graph type. Data requirements can be any of the following kinds:

- Geometric: Some graph types need a certain number of data points in order to display data. For example, a line graph requires at least two groups of data because a line requires at least two points.
- Complex: Some graph types require more than one data point for each marker (which is the component that actually represents the data in a graph). A scatter graph, 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 graph does not have enough data points for each group, the graph component does its best to display a graph.
- Logical: Some graph types cannot accept certain kinds of data. The following examples apply:
  - Negative data issues: Do not pass negative data to a pie graph or to a percentage bar, line, or area graph. Markers will not display for negative data in percentage graphs.
  - Null or zero data: You cannot see markers for null data because markers will not be produced for null data. Also, if a graph receives zero data and the axis line is at zero, the marker is not visible. However, if the axis line is at nonzero, the zero marker is visible.
  - Insufficient sets (or series) of data: Dual-Y graphs require a set of data for each y-axis. Usually, each set represents different information. For example, the y1-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 graph cannot display data on two different Y-axes. It displays the data on a single y-axis.

Similar graphs share similar data requirements. For example, you can group the following graphs under the category of area graphs:

- Absolute area graph.
- Stacked area graph.
- Percentage area graph.

23.2.1 Graph Type Data Requirements

Specific data requirements for each graph type are defined as follows:

- Area graphs:
  - At least two groups of data is required for area graphs. A group is represented by a position along the horizontal axis that runs through all area markers. In a graph that shows data for a three-month period, the groups might be labeled Jan, Feb, and Mar.
  - One or more series of data is required for area graphs. A filled-in area represents a series or set of data and is labeled by legend text, such as the continent of the Americas, Europe, and Asia.
  - Percentage area graphs cannot have negative numbers.
  - Dual-Y graphs require two sets of data.
- Bar and horizontal bar graphs:
  - Percentage bar graphs cannot have negative numbers.
  - Dual-Y graphs require two sets of data.
- Bubble graphs:
  - At least three data values for a data marker are required. Each data marker in a bubble graph 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 graphs require that data must be in multiples of three. For example, in a specific bubble graph, 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 graph, 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 graphs:
  - Combination graphs require at least two sets of data or else the graph cannot show different marker types.
  - Combination graphs require at least two groups of data or else the graph cannot render an area marker or a line marker.

- Funnel graphs:
  - At least two series (or sets of data) are required for funnel graphs. These two sets of data serve as the target and actual data values. Threshold values appear in the graph legend.

    Another variation of the funnel graph requires only one set of data, where the data values shown are percentages of the total values. To produce this type of funnel graph, you must set the `funnelPercentMeasure` property on the graph to be `True`. This setting should be done in the XML for the graph.
  - At least one group of data is required to be used as a stage for funnel graphs.
- Line graphs:
  - At least two groups of data are required for line graphs because lines require at least two points. A group is represented by a marker of each color. The group has a tick label such as the name of a month.
  - Percentage line graphs cannot have negative numbers.
  - Dual-Y graphs require two sets of data.
- Pareto graphs:
  - At least two groups of data are required for Pareto graphs.
  - Pareto graphs cannot have negative numbers.
  - If you pass more than one set of data to a Pareto graph, the graph uses only the first set of data.
  - Do not pass percentage values as part of the data for a Pareto graph. The graph calculates the percentages based on the data that you pass.
- Pie and ring graphs:
  - At least one group of data is required for a pie or ring graph. The data structure is as follows:
    - Each pie or ring represents one group of data and has a pie or ring label such as the name of a month. If you have only one group of data, then only one pie or ring appears even if you selected a multiple pie or ring graph type. Also, if any group has all zero data, then the pie or ring for that group is not displayed.
    - A series or set of data is represented by all the slices of the same color. 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.
  - Pie graphs cannot have negative numbers.
  - Multiple pie graphs must have at least two sets of data.
- Polar graphs:

  Polar graphs are circular scatter graphs with a directional aspect. At least two data values are required for each marker in a polar graph. Each data marker represents the following:
  - The x value that determines the location of the marker along the x-axis, which is the location around the circle, clockwise.
  - The y value that determines the location of the marker along the y-axis, which is the distance from the center of the graph.

- Radar graphs:

  At least three groups of data are required for a radar graph. The data structure is as follows:

  – The number of sides on the polygon is equal to the number of groups of data. Each corner of the polygon represents a group.

  – A series or set of data is represented by a line, all the markers of the same color, or both. It is labeled by legend text.

- Scatter graphs:

  – At least two data values are required for each marker in a scatter graph. Scatter graphs have either a single y-axis or a dual y-axis. 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.

- Sparkcharts:

  – Sparkcharts do not accept tabular data or graphDataModel.

  – Line, bar, and area sparkcharts require a single series of data values.

  – Floating bar sparkcharts require two series of data values, one for the float offset, and one for the bar value

- Stock graphs:

  – Stock: High-Low-Close

    * Each stock marker requires a group of three data values in the following sequence: High, Low, Close. To display stock data for more than one day, data must be in multiples of three, such as three data values for Monday, three data values for Tuesday, and so on for each additional day.

    * A series (or set) of data is represented by markers of the same color that represent one stock. A series is labeled by legend text such as Stock A. The legend appears even if you have only one stock, with the exception of candle stock graphs. Most high-low-close stock graphs have only one series. If you show more than one series and the prices of the different stocks overlap, then some stock markers obscure other stock markers.

  – Stock: High-Low-Close with Volume:

    * Each stock marker requires a group of four data values in the following sequence: High, Low, Close, Volume. To display stock data for more than one day, data must be in multiples of four and sequenced as follows: Monday High, Monday Low, Monday Close, Monday Volume, and so on for each additional day.

    * High-low-close stock graphs that also show volume can display the data for only one stock. The label for this stock appears in the legend of the graph.

  – Stock: Open-High-Low-Close

    * Each stock marker requires a group of four data values in the following sequence: Open, High, Low, Close. To display stock data for more than one day, data must be in multiples of four, such as four data values for Monday, four data values for Tuesday, and so on.

    * A series (or set) of data is represented by markers that have the same color and represent one stock. A series is labeled by legend text such as Stock A. The legend appears even if you have only one stock. Most open-high-low-close stock graphs have only one series. If you show more than one series and the prices of the different stocks overlap, then some stock markers obscure other stock markers.

- Stock: Open-High-Low-Close with Volume
    * Each stock marker requires a group of five data values in the following sequence: Open, High, Low, Close, Volume. To display stock data for more than one day, data must be in multiples of five and sequenced as follows: Monday Open, Monday High, Monday Low, Monday Close, Monday Volume, and so on for each additional day.
    * Open-high-low-close stock graphs that also show volume can display the data for only one stock. The label for this stock appears in the legend of the graph.
- Candle: Open-Close
    * Each stock marker requires a group of two data values in the following sequence: Open, Close. To display stock data for more than one day, data must be in multiples of two, such as two data values for Monday, two data values for Tuesday, and so on.
    * A series (or set of data) is represented by markers for one stock. Candle stock graphs allow the display of values for only one stock. For this reason, no legend appears in these graphs and you should show the series label (which is the name of the stock) in the title of the graph.
- Candle: Open-Close with Volume
    * Each stock marker requires a group of three data values in the following sequence: Open, Close, Volume. To display stock data for more than one day, data must be in multiples of three, such as three data values for Monday, three data values for Tuesday, and so on.
    * A series (or set of data) is represented by markers for one stock. Candle stock graphs allow the display of values for only one stock. For this reason, no legend appears in these graphs and you should show the series label (which is the name of the stock) in the title of the graph.
- Candle: Open-High-Low-Close
    * Each stock marker requires a group of four data values in the following sequence: Open, High, Low, Close. To display stock data for more than one day, data must be in multiples of four, such as four data values for Monday, four data values for Tuesday, and so on.
    * A series (or set) of data is represented by markers for one stock. Candle stock graphs allow the display of values for only one stock. For this reason, no legend appears in these graphs and you should show the series label (which is the name of the stock) in the title of the graph.
- Candle: Open-High-Low-Close with Volume
    * Each stock marker requires a group of five data values in the following sequence: Open, High, Low, Close, Volume. To display stock data for more than one day, data must be in multiples of five, such as five data values for Monday, five data values for Tuesday, and so on.

- * A series (or set) of data is represented by markers for one stock. Candle stock graphs allow the display of values for only one stock. For this reason, no legend appears in these graphs and you should show the series label (which is the name of the stock) in the title of the graph.

23.2.2 Configuring Graphs

Because of the many graph types and the significant flexibility of the graph components, graphs have a large number of DVT tags. The prefix `dvt:` occurs at the beginning of each graph 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 graph component:

- Graph component tags: The 13 graph component tags provide a convenient and quick way to create a commonly used graph type. They are represented in the Components window as categories of graphs with one or more type variations.

  Table 23–1 provides a description of the graph component tags, and their variations as specified in the `subType` attribute of the graph component.

Table 23–1 Graph Component Tags and Sub Types

Graph Tag	Description	Sub Types

`areaGraph`	Represents data as a filled-in area.	`AREA_VERT_ABS`
		`AREA_VERT_ABS_SPLIT2Y`
		`AREA_VERT_PERCENT`
		`AREA_VERT_STACK`
		`AREA_VERT_STACK_SPLIT2Y`

`barGraph`	Represents data as a series of vertical bars.	`BAR_VERT_CLUST`
		`BAR_VERT_CLUST_SPLIT2Y`
		`BAR_VERT_CLUST2Y`
		`BAR_VERT_FLOAT_STACK`
		`BAR_VERT_PERCENT`
		`BAR_VERT_STACK`
		`BAR_VERT_STACK_SPLIT2Y`
		`BAR_VERT_STACK2Y`

`bubbleGraph`	Represents data by the location and size of round data markers (bubbles).	`BUBBLE` `BUBBLE_2Y`
`comboGraph`	Uses different types of data markers (bars, lines, or areas) to display different kinds of data items.	`COMBINATION_VERT_ABS` `COMBINATION_VERT_ABS_2Y`
`funnelGraph`	Visually represents data related to steps in a process. The steps appear as vertical slices across a horizontal cone-shaped section.	`FUNNEL`

`horizontalBarGraph`	Displays data as bars horizontally along the y-axis.	`BAR_HORIZ_CLUST`
		`BAR_HORIZ_CLUST_SPLIT2Y`
		`BAR_HORIZ_CLUST2Y`
		`BAR_HORIZ_PERCENT`
		`BAR_HORIZ_STACK`
		`BAR_HORIZ_STACK_SPLIT2Y`
		`BAR_HORIZ_STACK2Y`

Table 23–1 (Cont.) Graph Component Tags and Sub Types

Graph Tag	Description	Sub Types
lineGraph	Represents data as a line, as a series of data points, or as data points that are connected by a line.	LINE_VERT_ABS LINE_VERT_ABS_2Y LINE_VERT_ABS_SPLIT2Y LINE_VERT_PERCENT LINE_VERT_STACK LINE_VERT_STACK_SPLIT2Y LINE_VERT_STACK2Y
paretoGraph	Represents data by bars and a percentage line that indicates the cumulative percentage of bars.	PARETO
pieGraph	Represents one group of data as sections of a circle causing the circle to look like a sliced pie. Pie graphs can display as a pie or ring, where the center of each circle has a hole in which the total pie value is displayed.	PIE PIE_BAR PIE_MULTI RING RING_BAR RING_MULTI
radarGraph	Appears as a circular line graph.	RADAR_LINE
scatterGraph	Represents data by the location of data markers. A scatter graph can display data in a directional manner as a polar graph.	SCATTER SCATTER_2Y POLAR
sparkChart	Simple, condensed graph 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 floatingBar line
stockGraph	Displays stock prices and, optionally, the volume of trading for one or more stocks in a graph. When any stock or candle stock graph includes the volume of trading, the volume appears as bars in the lower part of the graph.	STOCK_CANDLE STOCK_CANDLE_VOLUME STOCK_HILO_CLOSE STOCK_HILO_CLOSE_VOLUME STOCK_OHLC_CANDLE STOCK_OHLC_CANDLE_VOLUME STOCK_OPEN_HILO_CLOSE STOCK_VOLUME

- Common graph child tags: These tags are supported by most graph component tag to provide a variety of customization options.

  Table 23–2 provides a list and description of these tags.

Table 23–2 Common Graph Child Tags

Child Tag	Description
animationOnDisplay animationOnDataChange	Configuring animation effects for graphs.

Table 23–2 (Cont.) Common Graph Child Tags

Child Tag	Description

`background`
`graphFont`
`graphFootnote`
`graphPlotArea`
`graphSubTitle`
`graphTitle`
	Customizing the appearance of graph elements including titles.

`attributeFormat`	Formatting categorical attributes in the ordinal axis and marker tooltips.

`legendArea`
`legendText`
`legendTitle`
	Customizing of the graph legend.

`markerText`
`x1Format`
`y1Format`
`y2Format`
`zFormat`
	Marker customization related to each axis.

`o1Axis`
`o1MajorTick`
`o1TickLabel`
`o1Title`
	Customizing the ordinal axis (also known as the category axis) used with bar, area, combination, line, radar, and stock graphs with group labels.

`x1Axis`
`x1MajorTick`
`x1TickLabel`
`x1MinorTick`
`x1Title`
	Customizing the x-axis used with scatter and bubble graphs with numerical labels.

`y1Axis`
`y1MajorTick`
`y1TickLabel`
`y1MinorTick`
`y1Title`
	Customizing the y1-axis.

- Child set tags: These tags wrap a set of an unlimited number of related tags.

  Table 23–3 provides a list and description of these tags.

Table 23–3 Graph Child Set Tags

Child Set Tag	Description
`alertSet`	Wraps `dvt:alert` tags that define an additional data point that needs to be highlighted with a separate symbol, such as an error or warning.
`annotationSet`	Wraps `dvt:annotation` tags that define an annotation on a graph. An annotation is associated with a specific data point on a graph
`referenceObjectSet`	Wraps `dvt:referenceObject` tags that define a reference line or a reference area for a graph. You can define an unlimited number of reference objects for a given graph.

Table 23–3 (Cont.) Graph Child Set Tags

Child Set Tag	Description
`seriesSet`	Wraps `dvt:series` tags that define a set of data markers or series on a graph.
`shapeAttributesSet`	Wraps `dvt:shapeAttributes` tags that specify interactivity properties on a subcomponent of a graph.

In each case, during design, you must create the wrapper tag first, followed by a related tag for each item in the set. Example 23–1 shows the sequence of the tags when you create a set of alert tags to define two alert points for an area graph.

Example 23–1 Code Sample for a Set of Alert Tags

```
<dvt:areaGraph id="areaGraph1" subType="AREA_VERT_ABS">
  <dvt:background>
    <dvt:specialEffects/>
  </dvt:background>
  <dvt:graphPlotArea/>
  <dvt:alertSet>
    <dvt:alert xValue="Boston" yValue="3.50"
               yValueAssignment="Y1AXIS" imageSource="myWarning.gif"/>
    <dvt:alert xValue="Boston" yValue="5.50"
               yValueAssignment="Y1AXIS" imageSource="myError.gif"/>
  </dvt:alertSet>
  <dvt:o1Axis/>
  <dvt:y1Axis/>
  <dvt:legendArea automaticPlacement="AP_NEVER"/>
</dvt:areaGraph>
```

- Graph-specific child tags: These tags apply either to specific graph types or to specific parts of a graph.

  Table 23–4 provides a list and description of these tags.

Table 23–4 Graph-Specific Child Tags

Child Tag	Description
`specialEffects` `gradientStopStyle`	Gradients that are used for a graph only in conjunction with `dvt:background`, `dvt:legendArea`, `dvt:graphPlotArea`, `dvt:graphPieFrame`, `dvt:series`, `dvt:referenceObject`, or `dvt:timeSelector` child components.

`sliceLabel`
`x1TickLabel`
`y2TickLabel`
`y1TickLabel`
`x1Format`
`y1Format`
`y2Format`
`zFormat`
`stockVolumeFormat`
	Formatting numerical data values for graph.

`timeAxisDateFormat` `timeSelector`	Time axis customization for area, bar, combination, line, and stacked bar graphs.
`timeSelector`	Selection of a range on a time axis for master-detail graphs

Table 23–4 (Cont.) Graph-Specific Child Tags

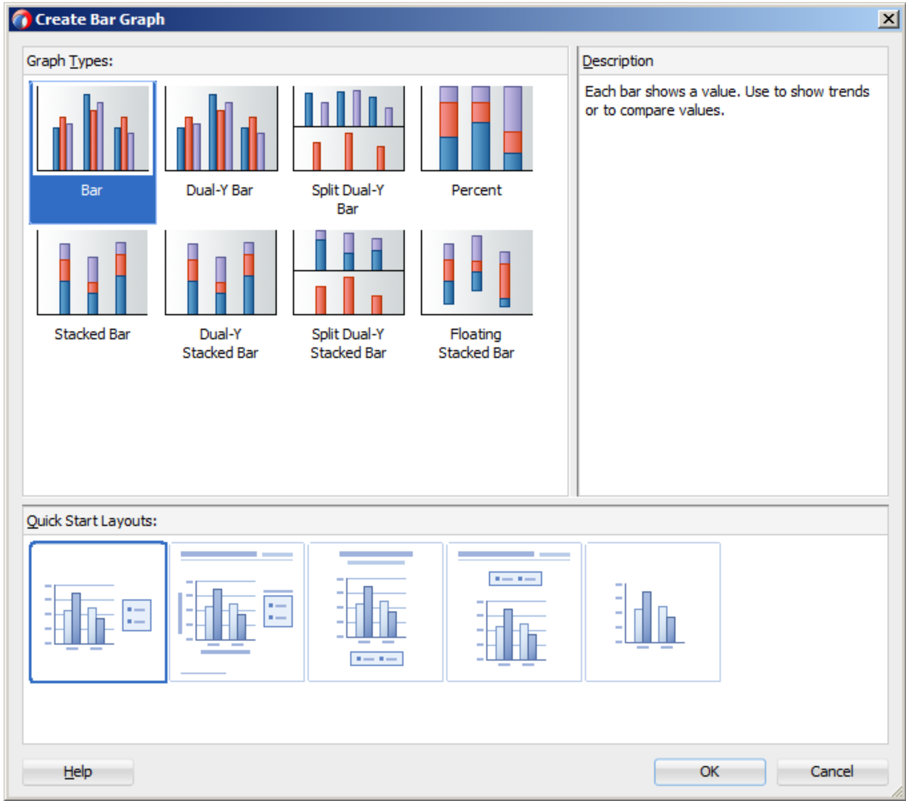
Child Tag	Description
paretoLine paretoMarker	Pareto graph customizations.
graphPieFrame slice sliceLabel	Pie graph customizations.
sparkItem	Sparkchart customizations.
stockMarker stockVolumeformat volumeMarker	Stock graph customizations.

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 graph tag in JDeveloper, select the tag in the Structure window and press F1 or click **Help**.

23.2.3 How to Add a Graph to a Page

When you are designing your page using simple UI-first development, you use the Components window to add a graph to a JSF page. When you drag and drop a graph component onto the page, a Create Graph dialog displays available categories of graph types, with descriptions, to provide visual assistance when creating graphs. You can also specify a quick start layout of the graph's title and legend. Figure 23–36 shows the Create Bar Graph dialog for bar graphs with the default vertical clustered bar graph type selected.

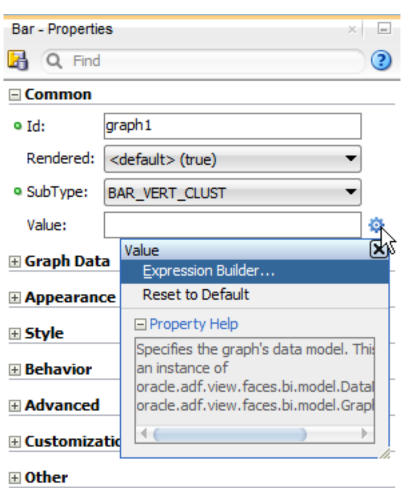
Figure 23–36 Create Bar Graph Dialog for Vertical Clustered Bar Graphs



Once you complete the dialog, and the graph is added to your page, you can use the Properties window to specify data values and configure additional display attributes for the graph.

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 23–37 shows the dropdown menu for a bar graph component `value` attribute.

Figure 23–37 Bar Graph Component Value Attribute Dropdown Menu



Note: If your application uses the Fusion technology stack, then you can use data controls to create a graph and the binding will be done for you. For more information, see the "Creating Databound Graphs" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

To add a graph to a page:

1. In the ADF Data Visualizations page of the Components window, from the Graph and Gauge panel, drag and drop the desired graph category onto the page to open the Create Graph dialog.
2. Use the dialog to select the graph type and the quick start layout for display of graph title, legend, and labels. For help with the dialog, press F1 or click **Help**.
3. Click **OK** to add the graph to your page.
4. In the Properties window, view the attributes for the graph. Use the help button to display the complete tag documentation for the graph type component.
5. Expand the **Common** section. Use this section to set the following attribute:
   - **SubType**: If you wish to change the variation of the graph type, select the desired type from the attribute dropdown menu. The valid values will vary depending on the graph.

     For example, the valid values for a bar graph are:

     – `BAR_VERT_CLUST`: Clustered bar graph that has a vertical orientation.

     – `BAR_VERT_CLUST_SPLIT2Y`: Clustered, vertical, split dual-y bar graph.

     – `BAR_VERT_CLUST2Y`: Clustered, vertical, dual-y bar graph.

     – `BAR_VERT_FLOAT_STACK`: Floating, vertical, stacked bar graph.

     – `BAR_VERT_PERCENT`: Percent, vertical bar graph.

     – `BAR_VERT_STACK`: Stacked, vertical bar graph.

     – `BAR_VERT_STACK_SPLIT2Y`: Stacked, vertical, split dual-y bar graph.

     – `BAR_VERT_STACK2Y`: Stacked, vertical, dual-y bar graph.
6. Expand the **Graph Data** section. Specify data values for the graph by setting the value in these fields:
   - **Value**: Specify the data model, which must be an instance of `DataModel`, using an EL Expression. Alternatively, set a metric value as either a `Java.lang.Number` object or a `String`.
   - **TabularValue**: Alternatively, specify a tabular data set as a `Java.util.List` object. For more information, see Section 23.2.5, "How to Create a Graph Using Tabular Data."

7. Expand the **Appearance** section. Specify display attributes by setting the value in these fields:

   - **ShortDesc**: Enter a statement of the graph's purpose and structure for use by screen readers
   - **EmptyText**: Specify the error text to display if the graph has no data.

The graph will display on the client in the HTML5 image format if the client supports it. For more information about graph image formats, see Section 23.2.6, "What You May Need to Know About Graph Image Formats."

23.2.4 What Happens When You Add a Graph to a Page

When a graph component is inserted into a JSF page using the Create Graph dialog, a set of child tags that support customization of the graph is automatically inserted. Example 23–2 shows the code inserted in the JSF page for a bar graph with the quick-start layout selected in Figure 23–36.

Example 23–2 Graph Sample Code

```
<dvt:barGraph id="graph1" subType="BAR_VERT_CLUST">
  <dvt:background>
    <dvt:specialEffects/>
  </dvt:background>
  <dvt:graphPlotArea/>
  <dvt:seriesSet>
    <dvt:series/>
  </dvt:seriesSet>
  <dvt:o1Axis/>
  <dvt:y1Axis/>
  <dvt:legendArea automaticPlacement="AP_NEVER"/>
</dvt:barGraph>
```

After inserting a graph component into the page, specialized context menus in the visual editor and Properties window buttons are available to support the customization of graph features. For more information, see Section 23.2.7, "Editing Graphs in the Visual Editor and Properties window."

23.2.5 How to Create a Graph Using Tabular Data

A graph is created when a grid of data is used for the graph component. The `tabularData` attribute of a graph component lets you specify a list of values that the graph uses to create a grid and to populate itself. To create a graph using tabular data you must store the data in a method in the graph's managed bean, and then use the graph component's `tabularData` attribute to reference the data.

For example, the table in Figure 23–38 has three columns: 2006, 2007, and 2008, and two rows: Shoes and Boots. This data produces a graph that compares annual sales for boots and shoes over a three-year period.

Figure 23–38 Comparison of Annual Sales

	2006	2007	2008
Shoes	120000	122000	175000
Boots	90000	110000	150000

In a managed bean, the list that contains the tabular data consists of a three-member `Object` array for each data value to be passed to the graph. The members of each array must be organized as follows:

- The first member (index 0) is the column label, in the grid, of the data value. This is generally a `String`. If the graph has a time axis, then this should be a `Java Date`. Column labels typically identify groups in the graph.
- The second member (index 1) is the row label, in the grid, of the data value. This is generally a `String`. Row labels appear as series labels in the graph, usually in the legend.
- The third member (index 2) is the data value, which is usually `Double`.

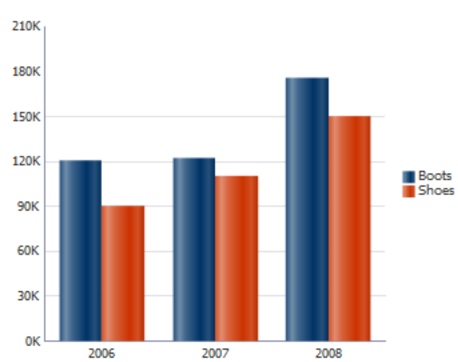
Example 23–3 shows code that creates the list of data required for a graph to compare annual sales of shoes and boots for a three-year period.

Example 23–3 Code to Create a List of Data for a Graph

```
public List getTabularData()
{
    ArrayList list = new ArrayList();
    String[] rowLabels  = new String[] {"Boots", "Shoes"};
    String[] colLabels  = new String[] {"2006", "2007", "2008"};
    Double [] [] values = new Double[][]{
        {120000.0, 122000.0, 175000.0},
        {90000.0, 110000.0, 150000.0}
        };
    for (int c = 0; c < colLabels.length; c++)
    {
     for (int r = 0; r < rowLabels.length; r++)
       {
        list.add (new Object [] {colLabels[c], rowLabels[r],
            new Double (values[r][c])});
       }
    }
    return list;
}
```

Figure 23–39 shows the graph that is rendered on the page if you add the method in Example 23–3 to a vertical clustered bar graph's `tabularData` attribute.

Figure 23–39 Bar Graph Using Tabular Data to Compare Annual Sales



Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

You will also need to create the method to supply data and add it to the graph's managed bean. If you need additional help, see Section 3.6, "Creating and Using Managed Beans."

To create a graph that uses data from a managed bean:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, expand the **Graph Data** section.
3. From the **TabularData** attribute dropdown menu, choose **Expression Builder**.
4. In the Expression Builder dialog, use the search box to locate the graph's managed bean.
5. Expand the managed bean node and select the method that contains the list of tabular data.
6. Click **OK**.

   The Expression is created.

   For example, for a managed bean named `sampleGraph` and a method named `getTabularData`, the Expression Builder generates the code `#{sampleGraph.tabularData}` as the value for the `tabularData` attribute of the graph.

23.2.6 What You May Need to Know About Graph Image Formats

Graphs support the following image formats: HTML5, Flash, and PNG. The image format used depends upon the application's settings and the client's environment.

You can configure your application to use a specific image format by setting or changing the following parameters:

- `oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT`

  This context initialization parameter is automatically added to `web.xml` for all new applications and defaults to HTML5. For more information, see Section A.2.3.28, "Graph and Gauge Image Format."

- Skin style

  Graphs will be displayed in the HTML5 image format when using the Skyros skin. New applications default to this skin. For more information about skinning and styles, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

- `flash-player-usage`

  You can disable 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 Section A.4.3, "Configuring Flash as Component Output Format."

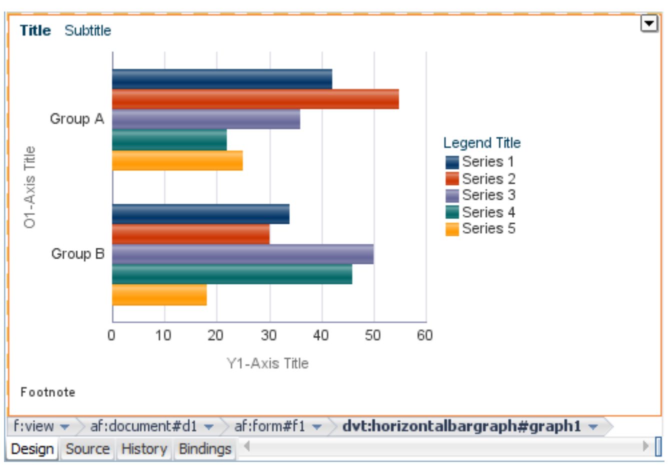
If the specified image format is not available on the client, the application will default to an available format. For example, if the client does not support HTML5, the application will use:

- Flash, if the Flash Player is available.
- Portable Network Graphics (PNG) output format. A PNG output format is also used when printing graphs. 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
  - Interactive pie slice behavior
  - Reference object hover behavior
  - Popup support
  - Selection
  - Series rollover behavior

23.2.7 Editing Graphs in the Visual Editor and Properties window

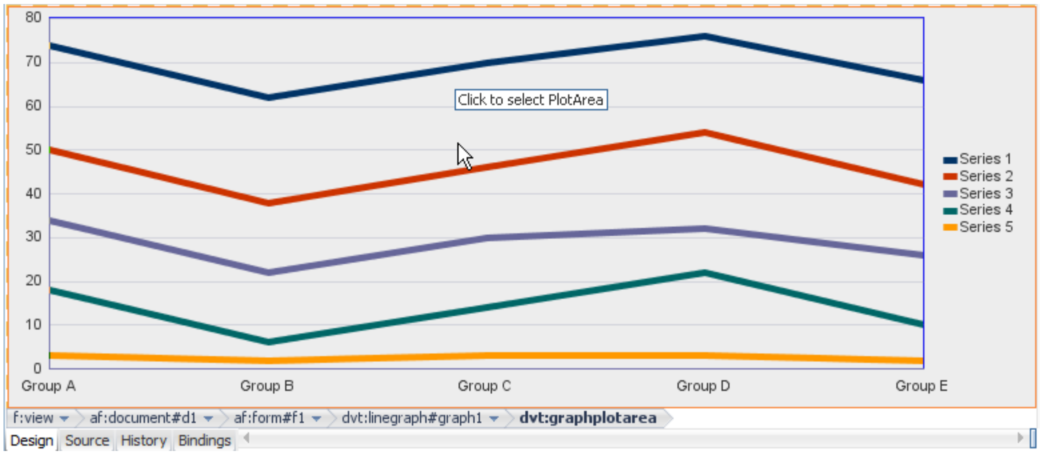
When you edit graph components in the visual editor and Properties window, specialized context menus and buttons are available to support the customization of graph features. Graph child components such as the title, legend area, plot area, background, axis labels, and display of graph series such as bars can be selected to display a context menu with editing choices. Figure 23–40 shows the display of a horizontal bar graph in the visual editor.

Figure 23–40 Horizontal Bar Graph in Visual Editor



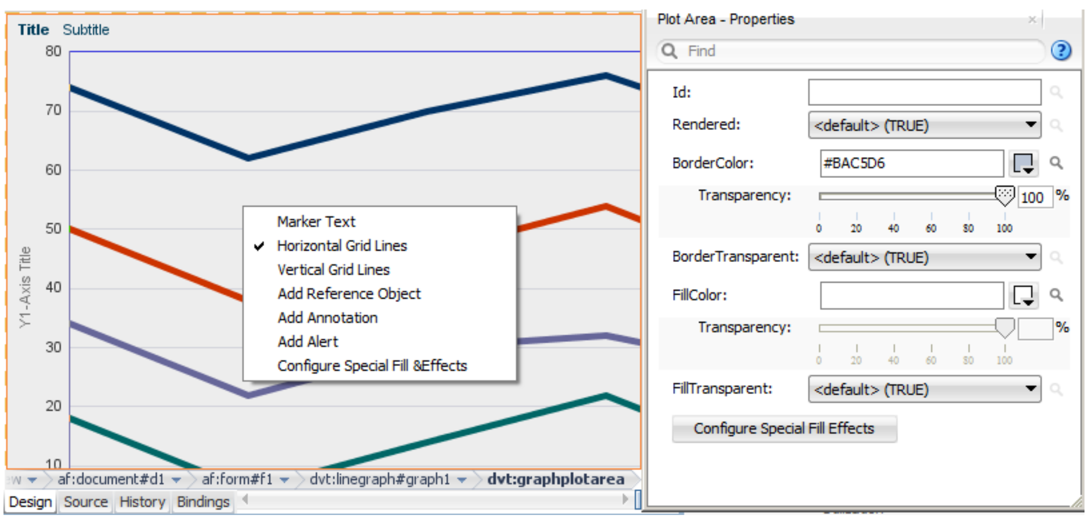
Popups in the visual editor provide confirmation of selection of the graph feature to be customized. For example, Figure 23–41 shows the popup displayed in the plot area of a line graph.

Figure 23–41 Visual Editor Popup in Line Graph



When the graph feature is selected in the visual editor, a specialized editing context menu is made available. Figure 23–42 shows the line graph plot area context menu from which you can choose a variety of options including removing the default display of the horizontal grid marks. You can also use the context menu or the Properties window buttons to configure special fill effects in the plot area. The selection of the graph tags is synchronized in the visual editor, Structure window, Properties window, and source editor.

Figure 23–42 Line Graph Plot Area Context Menu



For additional information about configuring line graphs, see. Section 23.5.3, "Changing the Appearance of Lines in Graphs." For additional information about configuring special fill effects, see Section 23.4.3, "Using Gradient Special Effects in Graphs."

23.3 Customizing Graph Display Elements

You can configure graph display elements including sizing, background and plot area appearance, title, axes, labels, legends, and tooltips.

23.3.1 Changing Graph Size and Style

You can customize the width and height of a graph and you can allow for dynamic resizing of a graph based on changes to the size of its container. You can also control the style sheet used by a graph. These two aspects of a graph are interrelated in that they share the use of the graph `inlineStyle` attribute.

23.3.1.1 How to Specify the Size of a Graph at Initial Display

You can specify the initial size of a graph by setting values for attributes of the `dvt:`*`type`*`Graph` tag. If you do not also provide for dynamic resizing of the graph, then the initial size becomes the only display size for the graph.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify the size of a graph at its initial display:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, in the **Style** section, enter a value for the **InlineStyle** attribute of the graph tag.

   For example, to create a graph that is 200 pixels in width and has a height of 200 pixels, use the following setting for the **InlineStyle** attribute: `width:200px;height:200px`.

23.3.1.2 How to Provide for Dynamic Resizing of a Graph

You must enter values in each of two attributes of the `dvt:`*`typeGraph`* tag to allow for a graph to resize when its container in a JSF page changes in size. The values that you specify for this capability also are useful for creating a graph component that fills an area across different browser window sizes.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To allow dynamic resizing of a graph:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, in the **Behavior** section, from the **DynamicResize** attribute's dropdown list, select the value `DYNAMIC_SIZE`.
3. In the **Style** section, for the **InlineStyle** attribute, enter a fixed number of pixels or a relative percent for both width and height.

   For example, to create a graph 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:** To specify a width of 100%, set the **StyleClass** to `AFStretchWidth`.

23.3.1.3 How to Use a Specific Style Sheet for a Graph

You have the option of selecting any of the standard styles available for the `dvt:`*`typeGraph`* tag. You can also specify a custom style sheet for use with a graph.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To select a style sheet for a graph:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, expand the **Style** section.
3. For the **StyleClass** attribute, enter the name of the CSS style class to use for the graph.
4. In the Property Editor, select the CSS style class to use for the graph.

   For example, select the `OraBGGrayLight` class to set a light gray background for the graph.

   For additional help with style sheets and CSS style classes, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

23.3.2 Changing Graph Background, Plot Area, and Title

The graph automatically provides default settings for its background and plot area based on the style it is using. You can customize these settings using child tags of the graph.

The graph also provides title, subtitle, and footnote options that you can specify. By default, no text is provided for titles and footnotes. When you enter this information, you can also specify the font and font characteristics that you want to use for the text.

23.3.2.1 How to Customize the Background and Plot Area of a Graph

You can customize the following parts of graphs related to background and plot area:

- Background: The area on which the graph is plotted.
- Plot area: A frame in which data is plotted for all graphs other than pie graphs. Axes are displayed on at least two borders of the plot area.
- Pie frame: A frame in which pie graphs are plotted without the use of axes.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To customize the background and plot area of a graph:

1. If you want to customize the background of a graph, do the following:
   a. In the Structure window, right-click the **dvt:background** node and choose **Go to Properties**.
   b. To change the background fill color, choose **Edit** from the **FillColor** attribute's dropdown and select the color to use in the Property Editor.
2. If you want to customize the plot area of any graph other than a pie graph, do the following:
   a. In the Structure window, right-click the **dvt:graphPlotArea** node and choose **Go to Properties**.
   b. In the Properties window, use the attribute dropdown menus to select colors that you want to customize for the plot area's **BorderColor** and **FillColor** attributes.
3. If you want to customize the plot area of a pie graph, do the following:
   a. In the Structure window, right-click the **dvt:graphPieFrame** node and choose **Go to Properties**.
   b. In the Properties window, use the attribute dropdown menus to select colors that you want to customize for the plot area's **BorderColor** and **FillColor** attributes.

Note: You can also customize the colors of the background and plot area in a graph by adding gradient special effects. For more information, see Section 23.4.3, "Using Gradient Special Effects in Graphs."

23.3.2.2 How to Specify Titles and Footnotes in a Graph

You have the option of specifying a title, subtitle, and footnote for a graph. You use a separate child tag of the graph for each of these text entries. The attributes of each of these child tags let you define the horizontal alignment of the text field, the text content, and whether or not the text should be rendered.

The tags for title, subtitle, and footnote support the use of a child graph font tag to let you identify the exact font characteristics to be used for each text field.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify titles and a footnote for a graph:

1. If you want to enter a graph title, do the following:
   a. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Title**.
   b. Use the Properties window to specify values in the attributes of the `dvt:graphTitle` tag.
   c. If you want to provide specific font characteristics for the text, then in the Structure window, right-click the **dvt:graphTitle** node and choose **Insert Inside Title** > **Font**.
   d. Use the Properties window to specify values for the attributes of the `dvt:graphFont` tag.
2. If you want to enter a graph subtitle, do the following:
   a. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Subtitle**.
   b. Use the Properties window to specify values in the attributes of the `dvt:graphSubtitle` tag.
   c. If you want to provide specific font characteristics for the text, in the Structure window, right-click the **dvt:graphSubtitle** node and choose **Insert Inside Subtitle** > **Font**.
   d. Use the Properties window to specify values for the attributes of the `dvt:graphFont` tag.
3. If you want to enter a graph footnote, do the following:
   a. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Footnote**.
   b. Use the Properties window to specify values in the attributes of the `dvt:graphFootnote` tag.
   c. If you want to provide specific font characteristics for the text, then in the Structure window, right-click the **dvt:graphFootnote** node and choose **Insert Inside Footnote** > **Font**.

d. Use the Properties window to specify values for the attributes of the `dvt:graphFont` tag.

23.3.3 How to Customize Graph Axes and Labels

Graphs can have the following axes:

- Ordinal axis (also known as the o1-axis): The ordinal (or category) axis of a graph shows ordered data, such as ratings or stages, or shows nominal data, such as different cities or different products. The ordinal axis appears on bar, line, area, combination, or radar graphs. When the ordinal axis is horizontal and contains time data, it is called a time axis.

  An example of an ordinal axis is the horizontal line across the bottom of the plot area of a vertical bar graph. The values along this axis do not identify the extent of the data shown. Instead, they identify the different groups to which the data belongs.

- x1-axis: The x1-axis shows the values that appear along the horizontal axis in a graph. This axis has regular intervals of numbers instead of group labels. It is referred to as the x-axis. The x1-axis appears on bubble and scatter graphs.
- y1-axis: The y1-axis is the primary y-axis. It is usually the vertical value axis along the left side of the plot area. It has regular intervals of numbers.
- y2-axis: The y2-axis is the secondary y-axis. It is usually the vertical axis along the right side of the plot area. It has regular intervals of numbers.

For each axis, there are several graph child tags that support customization. The following sections discuss the options available for various kinds of customization of an axis.

23.3.3.1 How to Specify the Title, Appearance, and Scaling of an Axis

The following graph child tags support customization of the title and appearance of an axis:

- Title: Specifies the text and alignment for an axis title. Includes the following tags: `dvt:o1Title`, `dvt:x1Title`, `dvt:y1Title`, and `dvt:y2Title`. An axis does not show a title unless you use the appropriate title tag.
- Axis: Controls the color, line width, scaling, increment between tick marks, visibility of the axis, and scrolling in specific graph types. Includes the following tags: `dvt:o1Axis`, `dvt:x1Axis`, `dvt:y1Axis`, `dvt:y2Axis`.

Note: Scaling attributes are not present on the `dvt:o1Axis` tag because the ordinal axis does not display numeric values.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a

Graph to a Page."

To specify the title and appearance of an axis:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > *AxisType* Title**.

   For example, to set the title for a bar graph's o1-axis, choose **Insert Inside Bar** > **ADF Data Visualizations** > **O1 Title**.

2. In the Properties window, enter the text for the axis title and optionally specify values for other attributes of this tag.

3. If you want to specify font characteristics for the title, do the following:

   a. In the Structure window, right-click the **dvt:*typeTitle*** node and choose **Insert Inside Title** > **Font**.

   b. In the Properties window, enter the desired values for the characteristics of the font.

23.3.3.2 Specifying Scrolling on an Axis

Scrolling on a graph axis can be specified for the following graph types:

- Area, bar, and line graphs for the `dvt:o1Axis`, `dvt:y1Axis`, and `dvt:y2Axis` tags.
- Bubble and scatter graphs for the `dvt:x1Axis`,`dvt:y1Axis`, and `dvt:y2Axis` tags.

By default, a graph axis `scrolling` attribute is set to `off`. You can specify these values for the `scrolling` attribute:

- `off`: Scrolling is disabled (default).
- `on`: Scrolling is enabled, and the scroll bar is always present.
- `asNeeded`: Scrolling is enabled, but the scrollbar is not initially present. After zooming on the graph, the scrollbar displays and remains visible for the session.
- `hidden`: Scrolling is enabled but the scroll bar is always hidden. User may use pan scrolling.

23.3.3.3 How to Control the Appearance of Tick Marks and Labels on an Axis

Tick marks are used to indicate specific values along a scale on a graph. The following graph child tags support customization of the tick marks and their labels on an axis:

- Major tick: Controls the color, width, and style of tick marks on the axis. Includes the following tags: `dvt:o1MajorTick`, `dvt:x1MajorTick`, `dvt:y1MajorTick`, and `dvt:y2MajorTick`. Major tick increments are calculated automatically by default, or you can specify the tick steps with the `majorIncrement` attribute.
- Minor tick: Controls the color, width, and style of minor tick marks on the axis. Includes the following tags: `dvt:x1MinorTick`, `dvt:y1MinorTick`, and `dvt:y2MinorTick`. Minor tick increments are one-half of the major tick increment by default, or you can specify the tick steps with the `minorIncrement` attribute. Minor ticks do not support labels.
- Tick label: Controls the rotation of major tick label text and lets you specify font characteristics for the label. Includes the following tags: `dvt:o1TickLabel`, `dvt:x1TickLabel`, `dvt:y1TickLabel`, and `dvt:y2TickLabel`. These tags can also have a `dvt:graphFont` child tag to change font characteristics of the label.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To control the appearance of the ordinal axis tick labels:

1. In the visual editor, select the **o1 Tick Label** element on the graph.

   Alternatively, you can select the **dvt:o1Axis** node in the Structure window, then in the Properties window click **Configure O1 Axis** and choose **Value Labels**.

2. In the Properties window, enter values as needed for the following properties:

   - **TextRotation**: Use to specify the degree of text rotation to improve readability of the tick labels.

     Tip: Use rotation angles that are multiples of 90 degrees to achieve best results. Other angles are not well supported across rendering technologies and are not recommended.

   - **TickLabelSkipMode**: Use to specify if and how tick labels will be displayed on the ordinal axis. When you set the value at `TLS_MANUAL`, you can optionally use the **tickLabelSkipCount** attribute to set the number of tick labels to display between tick labels and the **tickLabelSkipFirst** attribute to set the index of the first tick label to be skipped.

3. Optionally, in the Properties window, click the **Configure Font** button to set properties for the child `dvt:graphFont` tag.

To control the appearance of tick marks and labels on an x-axis:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > X1 Major Tick**.

2. In the Properties window, enter desired values for the attributes of this tag and click the **Configure Tick Label** button to add an **X1 Tick Label** tag to the graph.

3. In the Properties window, enter desired values for the **X1 Tick Label** and if desired, click the **Configure Font** button to specify font characteristics for the tick label.

4. If you want to specify minor ticks in the graph, do the following:

   a. In the Structure window, right-click the graph node and choose **Insert Inside *GraphType* > ADF Data Visualizations > X1 Minor Tick**.

   b. In the Properties window, enter desired values for the characteristics of the font.

   Note: For the `tickStyle` attribute you must specify a value other than `GS_NONE` or `GS_AUTOMATIC`.

The procedure for controlling the appearance of tick marks on any graph axis is similar to the procedure for the x-axis. However, you customize the major tick and label tags and insert the minor ticks related to the specific axis that you want to customize.

23.3.3.4 Formatting Numbers on an Axis

The `dvt:markerText` tag lets you to control the format of numbers on an axis. The following `dvt:markerText` child tags wrap the number format for specific axes: `dvt:x1Format`, `dvt:y1Format`, and `dvt:y2Format`.

Note: There is no format for the ordinal axis because that axis does not contain numeric values.

To format numbers on these axes, insert child tags for the appropriate axis as shown in Section 23.4.4, "Formatting Data Values in Graphs."

23.3.3.5 How to Set Minimum and Maximum Values on a Data Axis

The Y-axes have the following graph child tags to support the starting value of the axis: `dvt:y1Axis`, and `dvt:y2Axis`. You have the option of specifying different scaling on each y-axis in a dual y-axis graph. For example, the y1-axis might represent units in hundreds while the y2-axis might represent sales in thousands of dollars.

Some graphs, such as scatter and bubble graphs, contain a `dvt:x1Axis` child tag for which the minimum and maximum values can also be set.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify the starting value on a y1-axis:

1. In the Structure window, right-click the **dvt:y1Axis** node and choose **Go To Properties**.
2. In the Properties window, for the **AxisMinValue** field, enter the starting value for the y1-axis.
3. In the **AxisMinAutoScaled** field, select `false` from the attribute dropdown list.

   You must set this attribute in order for the minimum value to be honored.

To establish the starting value on a y2-axis, use a similar procedure, but first insert the `dvt:y2Axis` tag as a child of the graph.

23.3.4 How to Customize Graph Legends

Graph components provide child tags for the following kinds of customization for the legend:

- Specifying the color, border, visibility, positioning, and scrollability of the legend area relative to the graph, `dvt:legendArea` tag
- Specifying the font characteristics and positioning of the text that is related to each colored entry in the legend, `dvt:legendText` tag
- Specifying an optional title and font characteristics for the legend area, `dvt:legendTitle` tag

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To customize the legend area, legend text, and title:

1. In the Structure window, right-click the **dvt:legendArea** node and choose **Go to Properties**.
2. Use the Properties window to specify values for the attributes of this tag. For example, you can specify the following attributes for the legend area:
   - **AutomaticPlacement** and **Position**: Specify automatic positioning of the legend area on the right or the bottom of the graph with the default value of `AP_ALWAYS`. Setting the value at `AP_NEVER` requires the value of the `position` attribute to be used for positioning of the legend area.
   - **Scrolling**: Specify scrolling in the legend area when required space exceeds available space using the value `asNeeded`. By default the value is set to `off`.
   - **PositionHint**: For pie, radar, and polar graphs, specify the alignment of the legend toward the center of the plot area using the value `alignToCenter`. By default the value is set to `alignToEdge` which aligns the legend toward the edge of the graph frame.
   - **MaxWidth**: Specify the maximum width of the legend area as a percentage of the graph's area. By default the value is set to an empty string which automatically sets the width based on the graph's settings.

     For example, to set the maximum width of the legend to 50% of the graph's area, enter `50%`.
3. If you want to customize the legend text, do the following:
   a. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Legend Text**.
   b. Use the Properties window to enter values for the attributes of this tag.
   c. Right-click the **dvt:legendText** node and choose **Insert Inside Legend Text** > **Font**.
   d. Use the Properties window to specify values for the attributes of the font tag.
4. If you want to enter a title for the legend, do the following:

a. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Legend Title**.

b. Use the Properties window to enter values for the attributes of this tag.

c. Right-click the **dvt:legendTitle** node and choose **Insert Inside Legend Title** > **Font**.

d. Use the Properties window to specify values for the attributes of the font tag.

23.3.5 Customizing Tooltips in Graphs

Tooltips are useful to display identification or detail information for data markers. They can be very useful in smaller graphs without enough space to display `markerText`. Graphs automatically display tooltips for components like title, subtitle, footnote, legend text, and annotations when their text is truncated.

In most graphs, if you move the cursor over a data marker, then a tooltip is displayed. In a line or area graph, you must move the cursor over a data marker or at the corners of the line or area and not merely over the line or area.

You can specify that each graph marker (such as bars) displays a tooltip with information. The following graph attributes are used together to customize a graph tooltip:

- `markerTooltipType`: Specifies whether tooltips are displayed for markers (such as bars) and identifies the kind of information that appears in the tooltips. You have the option to display the following information: text only, values only, or text and values. For specific graph types, options include displaying cumulative data value for each stacked graph marker or displaying percentage data value for each pie slice marker.
- `seriesTooltipLabelType`: Specifies whether tooltips are displayed for each set of values that appear in a legend. This attribute also controls the kind of information that appears in a series tooltip. For example, you could choose to display text that identifies a general term for the entire series (such as Product) or a specific term for a given member of the series (such as a specific Product name).

> **Note:** The graph displays series tooltip labels only if the graph's `markerTooltipType` attribute has a setting that includes text.

- `groupTooltipLabelType`: Specifies whether tooltip labels are displayed for data groups along an axis. For example, sales for specific products might be grouped by years or quarters. You can choose to display text that identifies a general term for the entire group (such as Time) or specific terms for each member of the group (such as Q1, Q2, Q3, or Q4 for quarters).

You can quickly format all the marker tooltips in a graph by setting the graph's `markerTooltipTemplate` attribute to a tokenized String. The attribute accepts a String that may contain any number of a set of predefined tokens. For example:

```
<dvt:lineGraph markerTooltipTemplate="Template Based Tooltip NEW_LINE SERIES_LABEL
GROUP_LABEL NEW_LINE Value: Y_VALUE"/>
```

The list of supported tokens is described in Table 23–5.

Table 23–5 markerTooltipTemplate String Tokens

Token	Description
`NEW_LINE`	Inserts a new line.
`SERIES_LABEL`	The series label for the series of this marker.
`GROUP_LABEL`	The group label for the group of this marker.
`X_VALUE`	The X value of a scatter or bubble marker or continuous time axis marker.
`Y_VALUE`	The Y value of this marker (if this marker has a Y value).
`Z_VALUE`	The Z value (bubble size) of a bubble marker.
`PIE_VALUE`	The value of a pie slice.
`PIE_PERCENT`	The pie slice percentage value.
`ACTUAL_VALUE`	The actual value for a funnel slice.
`TARGET_VALUE`	The target value for a funnel slice.
`HIGH_VALUE`	The high value for a stock marker.
`LOW_VALUE`	The low value for a stock marker.
`CLOSE_VALUE`	The close value for a stock marker.
`OPEN_VALUE`	The open value for a stock marker.
`VOLUME_VALUE`	The volume value for a stock volume marker.
`CUM_VALUE`	The cumulative stacked value for a stacked graph.
`CUM_PERCENT`	The cumulative percentage value for a stacked percent graph or Pareto graph.

23.4 Formatting Graph Text, Colors, and Data Values

You can format the text, colors, and data values for all graph types.

23.4.1 Formatting Text in Graphs

You can format text in any of the following subcomponents of a graph:

- Annotation: Includes only the `dvt:annotation` tag.
- Axis title: Includes the `dvt:o1Title`, `dvt:x1Title`, `dvt:y1Title`, and `dvt:y2Title` tags.
- Axis tick label: Includes the `dvt:o1TickLabel`, `dvt:x1TickLabel`, `dvt:y1TickLabel`, and `dvt:y2TickLabel` tags.
- Graph title: Includes the `dvt:graphFootnote`, `dvt:graphSubtitle`, and `dvt:graphTitle` tags.
- Legend: Includes only the `dvt:legendText` tag.
- Marker: Includes only the `dvt:markerText` tag.

Use the `dvt:graphFont` tag as a child of the specific subcomponent for which you want to format text. For an example of formatting text in a graph, see Section 23.3.2.2, "How to Specify Titles and Footnotes in a Graph.".

23.4.1.1 How to Globally Set Graph Font Using a Skin

You can set the `font` attributes of graph 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 fonts used in graph 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 font styles for graph components. Graph component skin selectors include the following:

- `af|dvt-graphFootnote`
- `af|dvt-graphSubtitle`
- `af|dvt-graphTitle`
- `af|dvt-o1Title`
- `af|dvt-x1Title`
- `af|dvt-y1Title`
- `af|dvt-y2Title`
- `af|dvt-pieLabel`
- `af|dvt-ringTotalLabel`
- `af|dvt-legendTitle`
- `af|dvt-legendText`
- `af|dvt-markerText`
- `af|dvt-o1TickLabel`
- `af|dvt-x1TickLabel`
- `af|dvt-y1TickLabel`
- `af|dvt-y2TickLabel`
- `af|dvt-annotation`
- `af|dvt-sliceLabel`
- `af|dvt-tooltips`

You can also use ADF Data Visualization Tools global skin selectors to define the `font` attributes across multiple graph components. Global skin selector names end in the `:alias` pseudo-class, and affect the skin for more than one component. Global graph skin selectors include the following:

- `.AFDvtGraphFont:alias`: Specifies the `font` attributes for all graph components.
- `.AFDvtGraphTitlesFont:alias`: Specifies the `font` attributes for all graph title components.
- `.AFDvtGraphLabelsFont:alias`: Specifies the `font` attributes for all graph label components.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To use a custom skin to set graph fonts:

1. Add a custom skin to your application containing the defined skin style selectors for the graph subcomponents. You can specify values for the following attributes:

   - `-tr-font-family`: Specifies the font family (name). It may not contain more than one font. If multiple fonts are specified, the first font will be used.
   - `-tr-font-size`: Specifies the size of the font. Units of absolute size are defined as:
     - `pt`: Points - the standard font size used by CSS2, where 1 point equals 1/72nd of an inch.
     - `in`: Inches, where 1 inch equals 72 points.
     - `cm`: Centimeters, where 1 centimeter equals approximately 28 points.
     - `mm`: Millimeters, where 1 millimeter equals approximately 2.8 points.
     - pc: Picas, where 1 pica equals 12 points.

     You can also use font size names for this attribute, including the following:

     - `xx-small`: 8 points
     - `x-small`: 9 points
     - `small`: 10 points
     - `medium`: 12 points
     - `large`: 14 points
     - `x-large`: 16 points
     - `xx-large`: 18 points
   - `-tr-font-style`: Specifies the style of the font. Valid values are `normal` or `italic`.
   - `-tr-font-weight`: Specifies the weight of the font. Valid values are `normal` or `bold`.
   - `-tr-text-decoration`: Specifies whether or not the underline emphasis is rendered. Valid values are `none` or `underline`.
   - `-tr-color:` Specifies the color of the font. Valid values are color names for HTML and CSS. The World Wide Consortium lists 17 valid color names including `aqua`, `black`, `blue`, `fuchsia`, `gray`, `green`, `lime`, `maroon`, `navy`, `olive`, `orange` (CSS 2.1), `purple`, `red`, `silver`, `teal`, `white`, and `yellow`.

     You can also use 3, 6, or 8 digits HEX (alpha channel is first 2 digit in 8 digit HEX), RGB, or RGBA.

   For example, specify the font family for all graph titles in a `mySkin.css` file as follows:

```
af|dvt-graphTitle
{
```

```
  -tr-font-family: Comic Sans MS;
}
```

For help with creating a custom skin, see the "Creating an ADF Skin File" section in *Creating ADF Skins with Oracle ADF Skin Editor*.

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 *Creating ADF Skins with Oracle ADF Skin Editor*.

For additional information about using styles and skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

23.4.2 Specifying Transparent Colors for Parts of a Graph

You can specify that various parts of a graph show transparent colors by setting the `borderTransparent` and `fillTransparent` attributes on the graph child tags related to these parts of the graph. By default, these attributes are set to `true`. The following list identifies the parts of the graph that support transparency:

- Graph background: Use the `dvt:background` tag. This element does not contain a border, and the `borderTransparent` attribute does not apply.
- Graph legend area: Use the `dvt:legendArea` tag.
- Graph pie frame: Use the `dvt:graphPieFrame` tag.
- Graph plot area: Use the `dvt:graphPlotArea` tag.

23.4.3 Using Gradient Special Effects in Graphs

A *gradient* is a special effect in which an object changes color gradually. Each color in a gradient is represented by a stop. The first stop is stop 0, the second is stop 1, and so on. You can specify any number of stops in the special effects for a subcomponent of a graph that supports special effects.

You can define gradient special effects for the following subcomponents of a graph:

- Graph background: Use the `dvt:background` tag.
- Graph plot area: Use the `dvt:graphPlotArea` tag.
- Graph pie frame: Use the `dvt:graphPieFrame` tag.
- Legend area: Use the `dvt:legendArea` tag.
- Series: Use the `dvt:series` tag.

> **Note:** By default, a graph's series gradient is set in the `seriesEffect` attribute with a value of `SE_AUTO_GRADIENT` to make the data markers appear smoother and apply graphic antialiasing. You must set the attribute to `SE_NONE` in order to specify a custom series gradient.

- Time selector: Use the `dvt:timeSelector` tag.
- Reference area: Use the `dvt:referenceObject` tag.

The approach that you use to define gradient special effects is identical for each part of the graph that supports these effects.

23.4.3.1 How to Add Gradient Special Effects to a Graph

For each subcomponent of a graph to which you want to add special effects, you must insert a `dvt:specialEffects` tag as a child tag of the subcomponent if it doesn't already exist. For example, if you want to add a gradient to the plot area of a graph, then you would create one `dvt:specialEffects` tag that is a child of the `dvt:graphPlotArea` tag.

Then, optionally if you want to control the rate of change for the fill color of the subcomponent, you would insert as many `dvt:gradientStopStyle` tags as you need to control the color and rate of change for the fill color of the component. These `dvt:gradientStopStyle` tags then must be inserted as child tags of the single `dvt:specialEffects` tag.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To add a gradient special effect to a graph:

1. In the Structure window, locate the graph's child node for the component that will contain the gradient special effect and expand if needed.

   For example, to set a gradient special effect on the graph's plot area, locate the **dvt:graphPlotArea** node and expand if needed.

2. If the selected child node does not contain a **dvt:specialEffects** child node, right-click the node and choose **Insert Inside *ChildNode* > Special Effects**.

   For example, to set a gradient special effect on the graph's plot area, right-click **dvt:graphPlotArea** and choose **Insert Inside Plot Area > Special Effects**.

3. Use the Properties window to enter values for the attributes of the `dvt:specialEffects` tag:

   a. **FillType**: Choose `FT_GRADIENT`.

      For **GradientDirection**, select the direction of change that you want to use for the gradient fill.

   b. **NumStops**: Enter the number of stops to use for the gradient.

4. Optionally, in the Structure window, right-click the **dvt:specialEffects** node and choose **Insert Inside Special Effects > Gradient Stop Style** if you want to control the color and rate of change for each gradient stop.

5. Use the Properties window to enter values for the attributes of the `dvt:gradientStopStyle` tag:

   a. **StopIndex**: Enter a zero-based integer as an index within the `dvt:gradientStopStyle` tags that are included within the `specialEffects` tag.

   b. **GradientStopColor**: Enter the color that you want to use at this specific point along the gradient.

c. **GradientStopPosition**: Enter the proportional distance along a gradient for the identified stop color. The gradient is scaled from 0 to 100. If 0 or 100 is not specified, default positions are used for those points.

6. Repeat Step 4 and Step 5 for each gradient stop that you want to specify.

23.4.3.2 What Happens When You Add a Gradient Special Effect to a Graph

Example 23–4 shows the XML code that is generated when you add a gradient fill to the background of a graph and specify two stops.

Example 23–4 XML Code Generated for Adding a Gradient to the Background of a Graph

```
<dvt:graph>
  <dvt:background borderColor="#848284">
    <dvt:specialEffects fillType="FT_GRADIENT" gradientDirection="GD_RADIAL"
                        gradientNumStops="2">
      <dvt:gradientStopStyle stopIndex="0" gradientStopPosition="60"
                             gradientStopColor="FFFFCC"/>
      <dvt:gradientStopStyle stopIndex="1" gradientStopPosition="90"
                             gradientStopColor="FFFF99"/>
    </dvt:specialEffects>
  </dvt:background>
</dvt:graph>
```

23.4.4 Formatting Data Values in Graphs

The attributes in a data collection can be data values or categories of data values. Data values are numbers represented by markers, like bar height, or points in a scatter graph. Categories of data values are members represented as an ordinal axis label or appear as additional properties in a tooltip. You can format both numerical and categorical attributes by using ADF Faces converter tags, including `af:convertNumber` for numerical data values, and `af:convertNumber`, `af:convertDateTime`, and `af:convertColor` for categorical data values.

Converter tag attributes let you format percents, scale numbers, control the number of decimal places, placement of signs, and so on. For more information about ADF Faces converters, see Chapter 7, "Validating and Converting Input."

23.4.4.1 How to Format Categorical Data Values

Categorical data values in graphs are represented by the name in the page definition file (`<pagename>PageDef.xml`) that defines the graph's data model. Example 23–5 shows the XML code in a page definition file for a page with a graph displaying categorical data values for the hire date and the bonus cost for employees.

Example 23–5 Categorical Data Value Names in Page Definition File

```
<graph IterBinding="EmpView1Iterator" id="EmpView1"
       xmlns="http://xmlns.oracle.com/adfm/dvt" type="BAR_VERT_CLUST">
  <graphDataMap leafOnly="true">
    <series>
      <data>
        <item value="Bonus"/>
      </data>
    </series>
    <groups>
        <item value="Hiredate"/>
    </groups>
  </graphDataMap>
```

```
</graph>
```

For each categorical attribute to be formatted, use the `dvt:attributeFormat` tag to specify the name of the categorical data value, and specify the child converter tag to be used when formatting the attribute. You can use `af:convertNumber`, `af:convertDateTime`, and `af:convertColor` to specify formatting for a categorical attribute.

For example, you can format the Hiredate and Bonus categorical data values defined in the page definition file in (`<pagename>PageDef.xml`).

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To format categorical data values defined in the page definition file:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Attribute Format**.
2. In the Properties window, enter the information for the **Name** attribute.

   For example, to specify a value for the hire date in Example 23–5, enter `Hiredate` for the **Name** attribute.
3. In the Structure window, right-click the attribute format tag you named and choose **Insert Inside Attribute Format > Convert *Type***.

   For example, to continue formatting `Hiredate`, right-click the **dvt:attributeFormat** node and choose **Insert Inside Attribute Format** > **Convert Date Time**.
4. In the Structure window, right-click the **af:*convertType*** node and choose **Go to Properties**.
5. Use the Properties window to enter values for the converter. For additional help, see Chapter 7, "Validating and Converting Input."
6. Repeat Step 1 through Step 5 for each additional attribute.

   For example, to complete the formatting for categorical data values in Example 23–5, repeat Step 1 through Step 5, setting `Bonus` as the name of the attribute, adding an `af:convertNumber` converter, and formatting the attribute for currency.

Example 23–6 shows the XML code that is generated if you format the categorical data values in a bar graph.

Example 23–6 Formatting Categorical Data Values in a Bar Graph

```
<dvt:barGraph id="barGraph1" value="#{bindings.EmpView1.graphModel}"
              subType="BAR_VERT_CLUST">
  <dvt:attributeFormat id="af1" name="Hiredate">
    <af:convertDateTime pattern = "yyyy-MM-dd hh:mm:ss a"  timeZone="US/Pacific"/>
  </dvt:attributeFormat>
  <dvt:attributeFormat id="af2" name="Bonus">
```

```
    <af:convertNumber type = "currency" currencySymbol = "$"
  </dvt:attributeFormat>
</dvt:barGraph
```

Note: If there is a single categorical date attribute being displayed on the ordinal (o1) axis, then the graph displays a time axis. The time axis will show dates in a hierarchical format as opposed to a single label on the axis, for example, June 27, 2001. To display a single label on the ordinal axis, the time axis should be turned off, for example `timeAxisType="TAT_OFF"`, and a `dvt:attributeFormat` tag should be used to specify the date format.

23.4.4.2 How to Format Numerical Data Values

Use the ADF Faces `af:convertNumber` tag to specify formatting for numeric data values related to any of the following `graph` tags:

- `dvt:sliceLabel`
- `dvt:stockVolumeFormat`
- `dvt:x1TickLabel`
- `dvt:x1Format`
- `dvt:y1TickLabel`
- `dvt:y1Format`
- `dvt:y2TickLabel`
- `dvt:y2Format`
- `dvt:zFormat`

For example, by default a pie graph shows the relationship of parts to a whole, represented as slices in a pie, and each slice label displays the percentage that each slice represents. You can configure a pie graph to represent each slice as a value such as the cost of materials, labor, and profit that make up the list price. You specify the `textType` attribute of the `dvt:sliceLabel` tag to display the value represented in the slice, and format the number accordingly.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a pie graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To format numbers in the slice label of a pie graph:

1. In the Structure window, right-click the **dvt:sliceLabel** node and choose **Go to Properties**.
2. In the Properties window, choose `LD_VALUE` from the **TextType** attribute dropdown list to specify that the pie slice in the graph represents a value.

3. In the Properties window, click **Configure Slice Label** and choose **Number Format** from the dropdown menu.
4. In the Properties window, choose `currency` from the **Type** attribute's dropdown list to specify the values as currency, and enter a dollar sign (`$`) in the **CurrencySymbol** attribute to use a dollar sign as the currency symbol.

Example 23–7 shows the XML code that is generated if you format the numbers in the slice label of a graph to appear as currency, and use the dollar sign symbol.

Example 23–7 Formatting the Numbers in the Slice Label of a Pie Graph

```
<pieGraph>
...
  <dvt:sliceLabel textType="LD_VALUE">
    <af:convertNumber type="currency" currencySymbol="$"/>
  </dvt:sliceLabel>
...
</pieGraph>
```

You can also use the ADF Faces `af:convertNumber` tag to format numbers in the marker text of a graph.

For example, you can provide different formatting for the marker text of each axis in the graph. In this procedure, the `af:convertNumber` tag is used to format the marker text on `dvt:y1axis`.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To format numerical data values for the marker text associated with the y1-axis of a graph:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Marker Text**.
2. In the Properties window, optionally enter values for attributes of `dvt:markerText`. For example, select `true` for the **Rendered** attribute to display the text in the graph.
3. In the Properties window, click **Configure Marker** and choose **Y1 Format**.
4. In the Properties window, optionally enter values as needed for the `dvt:y1Format` attributes.
5. In the Properties window, click **Configure Number Format** and specify values as needed for the attributes of the `af:convertNumber` tag. For example, select a `percent` value for the **Type** attribute to place a percentage sign after the marker text.

Example 23–8 shows the XML code that is generated when you format the numbers in the marker text for the y1-axis of a graph. This example specifies that numbers are followed by a percentage sign and that the text appears above the markers. For example, in a bar graph, the text will appear above the bars.

Example 23–8 Formatting Numbers in Graph Marker Text

```
<dvt:barGraph>
  <dvt:markerText rendered="true" markerTextPlace="MTP_OUTSIDE_MAX">
    <dvt:y1Format>
      <af:convertNumber type="percent"/>
    </dvt:y1Format>
  </dvt:markerText>
</dvt:barGraph>
```

Note: When the **textType** attribute of a pie slice label is set to percent (LD_PERCENT), or the **markerTooltipType** attribute of a graph tooltip is set to percent (MTT_PERCENT_*XXX*), a child `af:convertNumber` tag, if used, will be automatically set to `percent` for its **type** attribute. When `af:convertNumber` is forced to `percent`, graph clears the pattern attribute. This means that patterns are ignored when a graph forces percent formatting. This is applicable for pie, Pareto, funnel and any bar, line, or area percent graph.

23.4.4.3 What You May Need to Know About Automatic Scaling and Precision

In order to achieve a compact and clean display, graphs automatically determine the scale and precision of the values being displayed in axis labels, marker text, and tooltips. For example, a value of 40,000 will be formatted as 40K, and 0.230546 will be displayed with 2 decimal points as 0.23.

Automatic formatting still occurs when `af:convertNumber` is specified. Graph tags that support `af:convertNumber` child tags have `scaling` and `autoPrecision` attributes that can be used to control the graph's automatic number formatting. By default, these attribute values are set to `scaling="auto"` and `autoPrecision="on"`. Fraction digit settings specified in `af:convertNumber`, such as `minFractionDigits`, `maxFractionDigits`, or `pattern`, are ignored unless `autoPrecision` is set to `off`.

23.5 Customizing the Appearance of Series and Groups of Data

You can customize the appearance of series and groups of data for color, style, display. You can also customize the appearance of lines in a line graphs, pie slice in a pie graph, and markers in bubble and scatter graphs. You can also add reference lines to a graph.

23.5.1 Changing the Color, Style, and Display of Graph Data Values

For most graph types, an entry appears in the legend for each set of data values represented as graph bars, lines, areas, points, and slices. This entry identifies a set of related data values and displays the color that represents the set in the graph. For example, a bar graph might use yellow bars to represent the sales of shoes and green bars to represent the sales of boots. The graph component refers to each set of related data values as a *series*.

The graph automatically assigns a different color to each set of data values. You can customize the colors assigned to each series, including the fill color and the border color. For some graph types, you can enable filtering the display of data values in a graph by hiding or showing the series from the graph legend.

You can specify additional characteristics for specific graph types such as the width and style of lines in a line graph with choices including solid lines, dotted lines, lines with dashes, and so on. For more information, see Section 23.5.3, "Changing the Appearance of Lines in Graphs."

For scatter graphs you can separate data marker shape and color from the series to display the interdependence of data values. For more information, see Section 23.5.5, "Customizing Scatter Graph Series Marker Data Values."

You can also customize the colors of each series in a graph by adding gradient special effects. For more information, see Section 23.4.3, "Using Gradient Special Effects in Graphs."

23.5.1.1 How to Specify the Color and Style for Individual Series Items

Use one `dvt:seriesSet` tag to wrap all the individual `dvt:series` tags for a graph and set attributes for color and style of graph data markers.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify the color and style for series items in a graph:

1. In the Structure window, right-click the **dvt:seriesSet** child tag in the graph node, and choose **Go to Properties**.
2. Optionally, use the Properties window to specify values for attributes of the `dvt:seriesSet` tag.

   The attributes of this tag determine default settings for all series tags in the set. However, you can override these settings for a given series by entering values in the corresponding attributes of a `dvt:series` tag.
3. In the Structure window, expand the **dvt:seriesSet** node.
4. In the Structure window, right-click the **dvt:series node** and choose **Go to Properties**.

   The first `dvt:series` tag represents the first series item that appears in the Create Graph Binding dialog.
5. Use the Properties window to specify colors and other characteristics as needed for the `dvt:series` tag.
6. To configure additional series items, in the Structure window, right-click the **seriesSet** node and choose **Insert Inside Series Set** > **Series**.
7. Use the Properties window to specify colors and other characteristics as needed for the `dvt:series` tag.
8. Repeat Step 6 and Step 7 for each series item.

23.5.1.2 How to Enable Hiding and Showing Series Items

For graph types including area, bar, bubble, combination, line, pie, radar, and scatter, you can enable the hiding or showing of the series in a graph at runtime. Although at least one series must be displayed in the graph, users can filter the display of data values by clicking on the corresponding legend item.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To enable hiding and show series items:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, in the **Series** section of the **Appearance** attributes category, set the **HideAndShowBehavior** attribute of the graph. Valid values include:
   - `none`: Default value, no hide and show series behavior is enabled.
   - `withRescale`: Rescales the graph to show only the visible series.
   - `withoutRescale`: Hides the series, but does not rescale the graph.

23.5.2 Changing the Appearance of Pie Graphs

You can customize the appearance of pie graphs, and you can specify that you want one slice of a pie to be separated from the other slices in the pie.

23.5.2.1 How to Customize the Overall Appearance of Pie Graphs

You can customize the appearance of a pie graph by inserting any of the following child tags within the graph tag:

- `dvt:pieFeeler` tag: Specifies the color of a line, called a *pie feeler,* that extends from a pie slice to a slice label.
- `dvt:slice` tag: Specifies the location of a label for a pie slice.
- `dvt:sliceLabel` tag: Specifies the characteristics of the labels that describe each slice of a pie or ring graph. Each slice represents a data value. Use the `textType` attribute of this tag to indicate whether the slice label should show text only, value only, percent only, or text and percent. If you want to format numbers or specify font characteristics, you can add the following tags within the `dvt:sliceLabel` tag: `dvt:graphFont` and `af:convertNumber`.

You can also animate the pie slices which enables the users to click on slices to explode them or display a context menu with an option to expand or collapse all slices in the pie graph.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To customize the overall appearance of a pie graph:

1. To add a child tag to the pie graph, in the Structure window, right-click the **dvt:pieGraph** node and choose **Insert Inside Pie > ADF Data Visualizations > Pie Feeler** *or* **Slice** *or* **Slice Label**.
2. Use the Properties window to set values for the attributes as needed. For help with the properties, press F1 or click **Help**.
3. To animate the slices in a pie graph, in the Properties window, set the **InteractiveSliceBehavior** attribute on the `dvt:pieGraph` tag. Valid values are any combination of the following:
   - `none`: No interactive slice behavior enabled.
   - `explode`: User can click to explode the slices in a pie graph.
   - `explodeAll`: Add **Explode All** and **Unite All** options to a context menu.

   For example, you can specify that users can explode the slices in a pie graph and display a context menu to explode or collapse all the slices in the graph in the code:

   ```
   <dvt:pieGraph interactiveSliceBehavior="explode explodeAll"/>
   ```

   Note: The `interactiveSliceBehavior` attribute is only available in a Flash image format, while the `pieSliceExplode` attribute is available in all image formats.

23.5.2.2 How to Customize an Exploding Pie Slice

When one slice is separated from the other slices in a pie, this display is referred to as an *exploding pie slice*. The reason for separating one slice is to highlight that slice possibly as having the highest value of the quantity being measured in the graph.

The slices of a pie graph are the sets of data that are represented in the graph legend. As such, the slices are the series items of a pie graph.

Before you begin:

Follow the procedure in Section 23.5.1.1, "How to Specify the Color and Style for Individual Series Items" to create a series set that wraps individual series items.

To customize one slice in a pie graph:

1. In the Structure window, expand the **dvt:seriesSet** node and locate the **dvt:series** node that represents the pie slice that you want to separate from the pie.

2. Right-click the **dvt:series** node and choose **Go to Properties**.
3. In the Properties window, in the **Common** section, set the **PieSliceExplode** attribute between `0` and `100`, where 100 is the maximum exploded distance available.

23.5.3 Changing the Appearance of Lines in Graphs

You can use attributes of the `dvt:seriesSet` child of a graph tag to change the appearance of lines in graphs.

23.5.3.1 Displaying Either Data Lines or Markers in Graphs

You have the option of displaying data lines or data markers in a line, combination, or radar graph. If you display markers rather than data lines, then the markers appear in the legend automatically.

In the Properties window, set the following attributes of the `dvt:seriesSet` tag to display data lines or data markers:

- **LineDisplayed**: Specifies whether data lines appear in the graph. You can set these values:
  - `true` indicates that data lines are displayed in the graph.
  - `false` indicates that markers are displayed in the graph rather than data lines.
- **MarkerDisplayed**: Specifies whether markers or data lines appear in graph. You can set these values:
  - `true` indicates that markers are displayed in a graph.
  - `false` indicates that data lines are displayed in a graph.

> **Note:** Do not set both the `lineDisplayed` attribute and the `markerDisplayed` attribute to `false`.

23.5.3.2 Changing the Appearance of Lines in a Graph Series

You can customize the appearance of lines by using the `dvt:seriesSet` tag and the `dvt:series` tag as described in the following list:

- On the `dvt:seriesSet` tag, you can affect all the `dvt:series` tags within that set by specifying values for the following attributes:
  - `defaultMarkerShape`: Used only for line, scatter, polar, bubble, and combination graphs. Identifies a default marker shape for all the series in the series set.
  - `defaultMarkerType`: Used only for combination and line graphs. Valid values include `MT_AREA`, `MT_BAR`, `MT_MARKER`, and `MT_DEFAULT`.
- On the `dvt:series` tag, you can specify settings for each individual series using the following line attributes:
  - `lineWidth`: Specifies the width of a line in pixels
  - `lineStyle`: Specifies whether you want the graph to use solid lines, dotted lines, dashed lines, or dash-dot combination lines.

See the procedures in Section 23.5.1.1, "How to Specify the Color and Style for Individual Series Items" for more information about using the `dvt:seriesSet` tag and the `dvt:series` tag.

23.5.4 How to Customize Pareto Graphs

A Pareto graph identifies the sources of defects using a series of bars. The bars are arranged by value, from the greatest to the lowest number. The Pareto line shows the percentage of cumulative values of the bars, to the total values of all the bars in the graph. The line always ends at 100 percent.

You can customize the Pareto line and the Pareto marker by using the following graph child tags:

- `dvt:paretoLine` tag: Lets you specify the color, line width, and line style (such as solid, dashed, dotted, or a combination of dash-dot).
- `dvt:paretoMarker` tag: Lets you specify the shape of the Pareto markers.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a Pareto graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To customize a Pareto graph:

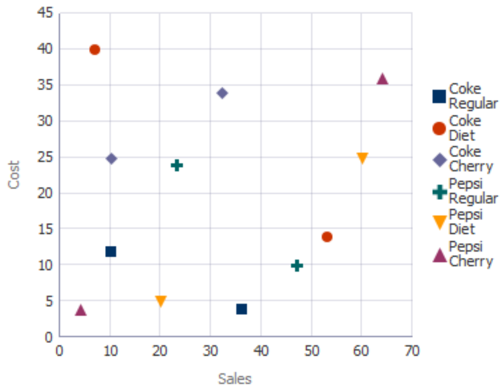
1. In the Structure window, right-click the **dvt:paretoLine** node and choose **Go to Properties**.
2. In the Properties window, specify values in the **Color**, **Width**, or **LineStyle** attributes.
3. In the Structure window, right-click the **dvt:paretoMarker** node and choose **Go to Properties**.
4. In the Properties window, select a value for the **MarkerShape** attribute.

23.5.5 Customizing Scatter Graph Series Marker Data Values

In scatter graphs, related data values in a series are represented by the data marker's shape and color. You can separate marker shape and color from the series to display the interdependence of data values.

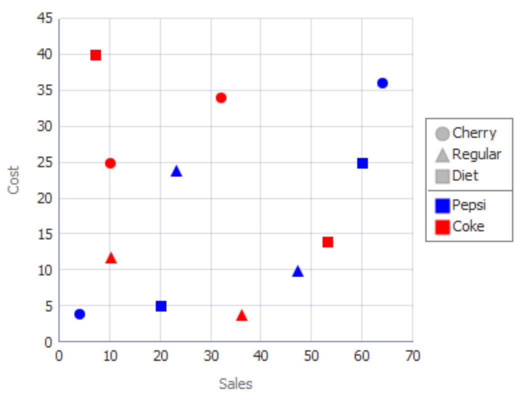
For example, Figure 23–43 shows a scatter graph that uses `Product` and `Brand` attributes collectively to determine the series represented by the data marker's shape and color.

Figure 23–43 Scatter Graph with Series Marker



The row header attributes can be used to override the default series specification. Figure 23–44 shows a scatter graph that displays the data values for the `Brand` attribute mapped to shapes and the `Product` attribute mapped to colors.

Figure 23–44 Scatter Graph with Series Item Markers



Use the following attributes to customize the scatter graph series marker data values:

- `markerShapeAttribute`: Specifies the row header attribute name to use to set the marker shape. The graph will display the default index based series marker shapes if this attribute is not specified.
- `markerColorAttribute`: Specifies the row header attribute name to use to set the marker color. The graph will display the default index based series marker colors if this attribute is not specified.

For example, specify `Product` and `Brand` as separate series item markers for a scatter graph using this code:

```
<dvt:scatterGraph id="scatter" markerColorAttribute="Product"
markerShapeAttribute="Brand" value="#{bindings.View1.graphModel}"/>
```

You can also set the `markerShapeAttribute` and `markerColorAttribute` in the **Advanced** section of the Properties window for the **dvt:scatterGraph** node.

23.5.6 Customizing Graph Marker Shapes

The shape of line, scatter, polar, combination and bubble graph markers is specified in the graph `series` component `markerShape` attribute. By default, the line, scatter, polar or combination graph assigns marker shapes to each series rotating through square, circle, diamond, plus sign, and triangle, and the bubble graph assigns marker shapes to a circle. The default value for `markerShape` is `MS_AUTOMATIC`.

You can specify prebuilt graph marker shapes by setting the `series` component `markerShape` attribute to one of these values:

- `MS_NONE`: Do not use series markers
- `MS_AUTOMATIC`: Default setting. Use default markers.
- `MS_SQUARE`: Use to specify square markers.
- `MS_CIRCLE`: Use to specify circular markers.
- `MS_DIAMOND`: Use to specify diamond markers.
- `MS_PLUS`: Use to specify plus sign markers.
- `MS_TRIANGLE_DOWN`: Use to specify triangular markers with the point down.
- `MS_TRIANGLE_UP`: Use to specify triangular markers with the point up.
- `MS_HUMAN`: Use to specify human shaped markers. When the `markerSize` attribute is specified, the human marker is scaled so that its width matches the `markerSize` value.

For custom graph markers, the `shapePath` attribute can be used to specify the path of an SVG file that will get displayed in place of a predefined shape. For example:

```
<dvt:series shapePath="/resources/shapes/house.svg" />
```

Marker shapes can be also specified through CSS style properties in an ADF skin. Using graph style properties, predefined marker shapes can be overwritten, and the paths to SVG files for custom markers can be defined without using the `shapePath` attribute. When using style properties, the `shape` attribute in the `series` component is used for defining both predefined and custom shapes.

A predefined shape will be overwritten if a global or component-specific style property for that shape is specified in the ADF skin. For example, you can overwrite the predefined `circle` shape by specifying the `newCircle.svg` file in the graph component style property as follows:

```
af|dvt-graph::shape-circle{
  -tr-path: url(/resources/path/newCircle.svg);
}
```

In the JSF page, the `series` component `shape` attribute is set as follows:

```
<dvt:series id="s1" shape="circle"/>
```

To specify a custom shape in the `series` component `shape` attribute, you must use a prefix of `custom` in the shape style property name. For example, if the custom shape is named `customName`, then the ADF skin file should define either a global `.AFDVTShapeCustomName:alias` style property, or the graph specific `af|dvt-graph::shape-customName` with the `-tr-path` property pointing to the SVG file as follows:

```
af|dvt-graph::shape-customName{
  -tr-path: url(/resources/path/newCShape.svg);
}
```

In the JSF page, the marker component shape attribute is set as follows:

```
<dvt:series id="s1" shape="customName"/>
```

For information about using skins and style properties, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

23.5.7 Adding Reference Lines or Areas to Graphs

Reference lines and areas can be set to display always, on rollover only, or never, regardless of how many there are and whether they are associated with a series or an axis.

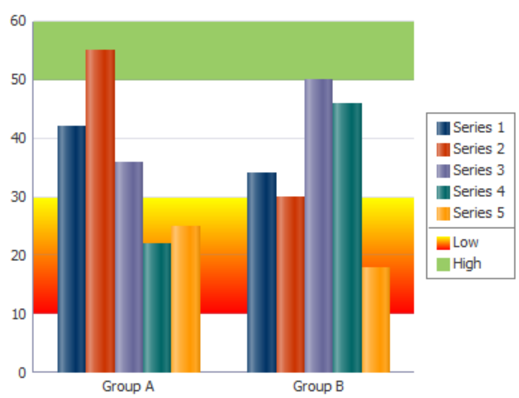
You can create reference lines that are associated with a series (that is a set of data values that appears as a single color in the graph legend). If there are multiple series with reference lines, then the reference lines show only when you move the cursor over a series marker or the corresponding series legend item. This is because multiple reference lines can be confusing to users.

You can also create reference areas that are associated with an axis. Typically, these areas are associated with a y-axis. If there are multiple reference areas, then these areas are also displayed when you move the cursor over the related axis.

If your application does not know how many reference lines or areas it will need until it is running, then you can create reference lines or areas dynamically at runtime.

For example, you could add areas to a bar graph to provide a reference for the values displayed in the graph. Figure 23–45 shows a bar graph with two reference areas for the high and low values of the graph.

Figure 23–45 Bar Graph with Reference Areas



23.5.7.1 How to Create Reference Lines or Areas During Design

Both reference lines and reference areas are created by the use of the following tags:

- `referenceObjectSet`: Wraps all the reference object tags for reference lines or reference areas for this graph.
- `referenceObject`: Identifies whether the tag represents a reference line or a reference area and specifies characteristics for the tag.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To add reference lines or areas to a graph during design:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Reference Object Set**.
2. Right-click the **dvt:referenceObjectSet** node and choose **Go to Properties**.
3. In the Properties window, if you are defining reference areas related to specific axes, then specify a value for displaying the appropriate axis or axes attributes: **displayX1**, **displayY1**, or **displayY2**.

   The value `RO_DISPLAY_AUTOMATIC` enables the display of a reference area only when the mouse moves over the related axis. This choice prevents the confusion that might occur if multiple reference areas were displayed all at the same time.

   Optionally, you can apply a gradient special effect to the reference area. For more information see Section 23.4.3, "Using Gradient Special Effects in Graphs."
4. If you are defining a reference line, specify a value for displaying the line.

   The value `RO_DISPLAY_AUTOMATIC` enables the display of a reference line only when the cursor moves over a series item (such as a bar) or over the corresponding series entry in the graph legend. This choice prevents the confusion that might occur if multiple series reference lines were displayed all the time.
5. In the Structure window, right-click the **dvt:referenceObjectSet** node and choose **Insert Inside Reference Object Set** > **Reference Object**.
6. Right-click the **dvt:referenceObject** node and choose **Go to Properties**.
7. In the Properties window, do the following:

   a. In the **Common** section, specify values for the **Index** attribute of the reference object, the **Type** attribute of the reference object (`RO_LINE` or `RO_AREA`), and the associated object in the **Association** attribute (a series for a reference line or a specific axis for a reference area). Also specify if the object should be displayed in the legend using the **DisplayedInLegend** attribute, and specify the text, if any, to display in the legend.

   b. If you are creating a reference line, then specify values for the attributes related to the line in the **Reference Line** section. This includes specifying the value of the line and series number of the series to which the line is related. The series number refers to the sequence in which the series appear in the Graph data binding dialog.

   c. If you are creating a reference area, then specify the low value and the high value in the **Reference Area** section that represent the reference area on the specified axis.

   d. Configure any additional attributes as needed.

      For example, use the **Color** attribute's dropdown menu to enter a color for the reference line or area. For additional help, press F1 or click **Help**.

23.5.7.2 What Happens When You Create Reference Lines or Areas During Design

When you create reference lines or areas during design, XML code is generated within the graph XML on the JSF page. The reference objects (both lines and areas) are wrapped by the `dvt:referenceObjectSet` tags. Example 23–9 shows the code for the two reference areas associated with the bar graph in Figure 23–45.

Example 23–9 XML Code for Reference Lines and Areas in a Graph

```
<dvt:barGraph shortDesc="Graph" id="bg1">
  <dvt:referenceObjectSet>
    <dvt:referenceObject type="RO_AREA" association="Y1AXIS"
                         location="RO_BACK" color="#55FFFF00"
                         lowValue="10" highValue="30"
                         displayedInLegend="true" text="Low">
      <dvt:specialEffects fillType="FT_GRADIENT"
                          gradientDirection="GD_DOWN"
                          gradientNumStops="2">
        <dvt:gradientStopStyle stopIndex="0" gradientStopPosition="0"
                               gradientStopColor="#FFFF00"/>
        <dvt:gradientStopStyle stopIndex="1"
                               gradientStopPosition="100"
                               gradientStopColor="#FF0000"/>
      </dvt:specialEffects>
    </dvt:referenceObject>
    <dvt:referenceObject type="RO_LINE" association="Y1AXIS"
                         location="RO_BACK" color="#99cc66"
                         lineValue="50"
                         displayedInLegend="true" text="High"/>
  </dvt:referenceObjectSet>
</dvt:barGraph>
```

23.5.7.3 How to Create Reference Lines or Areas Dynamically

If you want to create reference objects dynamically at runtime, then you use only the `referenceObjectSet` tag. You set the `referenceObjectMap` attribute on this tag with a method reference to the code that creates a map of the child component reference objects. The method that creates this map must be stored in a managed bean.

Example 23–10 shows sample code for creating a method to create a reference line dynamically.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To create reference lines or areas dynamically:

1. Create the managed bean to create the map of child component reference objects that you want to create dynamically during runtime. For additional help, see Section 3.6, "Creating and Using Managed Beans."

2. In the Structure window, right-click the graph node, then choose **Insert Inside *GraphType* > ADF Data Visualizations > Reference Object Set**.

3. Right-click the **dvt:referenceObjectSet** node and choose **Go to Properties**.

4. In the Properties window, specify in the **ReferenceObjectMap** attribute a method reference to the code that creates the map of child component reference objects.

   For example, for the managed bean `(sampleGraph)` and the method `getReferenceObjectMapList` shown in Example 23–10, the attribute should be set to the following value: `referenceObjectMap="#{sampleGraph.referenceObjectMapList}"`.

Example 23–10 Code for a Map of Child Reference Objects

```
Managed bean sampleGraph.java :
   public Map getReferenceObjectMapList() {
     HashMap map = new HashMap();
     ReferenceObject referenceObject = new ReferenceObject();
     referenceObject.setIndex(1);
     referenceObject.setColor(Color.red);
     referenceObject.setLineValue(30);
     referenceObject.setLineWidth(3);
     map.put(new Integer(1), referenceObject);
     return map;
    }
```

23.6 Animating Graphs

Graph components support animation effects such as slideshow transition for initial display of the graph component and for partial page refresh (PPR) events. Animation effects are specified in the graph's `animationOnDisplay` and `animationOnDataChange` properties with these values:

- `none` (default)
- `alphaFade`
- `conveyorFromLeft`
- `conveyorFromRight`
- `cubeToLeft`
- `cubeToRight`
- `flipLeft`
- `flipRight`
- `slideToLeft`
- `slideToRight`
- `transitionToLeft`
- `transitionToRight`
- `zoom`

Animation effects can also be performed using active data on graph types that support active data. The Active Data Service (ADS) allows you to bind ADF Faces components to an active data source using the ADF Model layer. To allow this, you must use the Fusion technology stack and configure the components and the bindings so that the components can display the data as it is updated in the source. Alternatively, you can configure the application to poll the data source for changes at prescribed intervals. For more information, see Section 23.6.1, "How to Configure Databound Graph Components to Display Active Data."

23.6.1 How to Configure Databound Graph Components to Display Active Data

The Active Data Service is supported for the following graph types:

Table 23–6 Graph Components Supporting Active Data

Graph Component	Graph Tag	SubType	3D Support?
Area	`dvt:areaGraph`	`AREA_VERT_ABS`	No
Bar	`dvt:barGraph`	`BAR_VERT_CLUST` (rectangular bar shape only)	Yes
		`BAR_VERT_STACK` (rectangular bar shape only)	
Bubble	`dvt:bubbleGraph`	`BUBBLE`	No
Combination	`dvt:comboGraph`	`COMBINATION_VERT_ABS` (rectangular bar shape only)	No
Line	`dvt:lineGraph`	`LINE_VERT_ABS`	No
Pie	`dvt:pieGraph`	`PIE`	Yes
Scatter	`dvt:scatterGraph`	`SCATTER`	No

To use the Active Data Service, you must have a data source that publishes events when data is changed, and you must create business services that react to those events and the associated data controls to represent those services. For more information about ADS and configuring your application, see Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

Configure databound graphs to display active data by setting a value on the binding element in the corresponding page definition file.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You will need to complete these tasks:

- Add a databound graph to your page. For more information, see Section 23.2.3, "How to Add a Graph to a Page."
- Create the data source that publishes events when data is changed and the associated business services and data controls. For more information, see Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

To configure a databound graph to display active data:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Page Definition** to open the page's associated page definition file.

2. In the Structure window for the page definition file, select the node that represents the attribute binding for the component.
3. In the Properties window, for the **ChangeEventPolicy** attribute, select `push`.

23.6.2 Specifying Animation Effects for Graphs

In the Properties window for the graph you wish to animate, set the following attributes:

- `animationOnDisplay`: Optional. Use to specify the type of initial rendering effect to apply. Valid values are:
  - none (default): Do not show any initial rendering effect.
  - `auto`: Apply an initial rendering effect automatically chosen based on graph type.
  - `alphaFade`
  - `conveyorFromLeft` or `conveyorFromRight`
  - `cubeToLeft` or `cubeToRight`
  - `flipLeft` or `flipRight`
  - `slideToLeft` or `slideToRight`
  - `transitionToLeft` or `transitionToRight`
  - `zoom`
- `animationOnDataChange`: Optional. Use to specify the type of data change animation to apply. Valid values vary, depending upon the graph type, and include:
  - `none`: Apply no data change animation effects.
  - `activeData` (default): Apply Active Data Service (ADS) data change animation events. This is the default value when the graph type supports active data.
  - auto: Apply partial page refresh (PPR) and ADS data change animation events.
  - `alphaFade`
  - `conveyorFromLeft` or `conveyorFromRight`
  - `cubeToLeft` or `cubeToRight`
  - `flipLeft` or `flipRight`
  - `slideToLeft` or `slideToRight`
  - `transitionToLeft` or `transitionToRight`
  - `zoom`
- `animationDuration`: Use to specify the animation duration in milliseconds.
- `animationIndicators`: Use to specify the type of data change indicators to show. Valid values are:
  - `none`: Show no data change indicators.
  - `all` (default): Show all data change indicators.
- `animationUpColor`: Use to specify the RGB hexadecimal color used to indicate that a data value has increased.

- `animationDownColor`: Use to specify the RGB hexadecimal color used to indicate that a data value has decreased.

23.7 Adding Interactive Features to Graphs

You can add interactive features to graphs including marker and legend dimming, zooming and scrolling, drilling, adding an interactive time axis, annotations and alerts, drag and drop, popups, selection support, and context menus.

23.7.1 How to Provide Marker and Legend Dimming

Markers include lines, bars, areas, scatter markers, bubbles, and pie slices. You can force all the data markers for a given set of data to be highlighted when you move the cursor over one data marker in the set or over the corresponding entry in the graph legend. The highlighting effect is visually achieved by dimming the other data markers in the set. For example, if a bar graph displays sales by month for four products (P1, P2, P3, P4), when you move the cursor over product P2 in January, all the P2 bars are highlighted, and the P1, P3, and P4 bars are dimmed.

Because the graph refers to all the data markers in a given set of data (such as all the P2 bars) as a series, the ability to highlight the data markers in a series is part of the graph's series rollover behavior feature.

Series rollover behavior is available only in the following graph types: bar, line, area, pie, scatter, polar, radar, and bubble graphs.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To dim all the data markers in a series:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the **Appearance** section, in the **SeriesRolloverBehavior** field, use the attribute dropdown list to select `RB_DIM`.

23.7.2 How to React to Changes in the Zoom and Scroll Levels

Zooming is enabled when you configure scrolling on one or more of the graph axes as described in Section 23.3.3.2, "Specifying Scrolling on an Axis."

You can provide custom code that will be executed when the zoom and scroll levels change on a graph. For example, you could write a method to determine which axis is zoomed, as well as the current extent of the zoomed axes.

You store the methods in a managed bean that takes as input a `ZoomEvent` or `ScrollEvent`. Example 23–11 shows sample code for creating methods to handle a graph's zoom and scroll events.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To provide custom behavior in response to zooming and scrolling in a graph:

1. Create a managed bean and add the methods that respond to zoom and scroll events. For additional help, see Section 3.6, "Creating and Using Managed Beans."

   For example, to use the code sample in Example 23–11, create a managed bean named `sampleGraph` and add the example methods to it.

2. In the Properties window, if not already enabled, configure scrolling on the axes that the methods will manage.

   For example, to use the sample code in Example 23–11, configure scrolling on the **dvt:o1Axis** node. For additional help with configuring scrolling, see Section 23.3.3.2, "Specifying Scrolling on an Axis."

3. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties.**

4. In the Properties window, expand the **Behavior** section and do one or both of the following:

   - In the **ZoomListener** field, specify a reference to the method that you stored in the managed bean.

     For example, if the method `setZoom` is stored in the managed bean `sampleGraph`, then the setting becomes: `"#{sampleGraph.setZoom}"`.

   - In the **ScrollListener** field, specify a reference to the method that you stored in the managed bean.

     For example, if the method `setScroll` is stored in the managed bean `sampleGraph`, then the setting becomes: `"#{sampleGraph.setScroll}"`.

Example 23–11 Sample Code to Set Zoom and Scroll

```
Managed bean sampleGraph.java:
public void setZoom(ZoomEvent event) {
        System.out.println("Start Group: " +
event.getAxisStartGroup(ZoomEvent.O1AXIS));
        System.out.println("Group Count: " +
event.getAxisGroupCount(ZoomEvent.O1AXIS));
        System.out.println("Start Group Label: " +
event.getAxisStartGroupLabel(ZoomEvent.O1AXIS));
}
public void setScroll(ScrollEvent event) {
        System.out.println("End Group Label: " +
event.getAxisEndGroupLabel(ScrollEvent.O1AXIS));
        System.out.println("Axis Min: " + event.getAxisMin(ScrollEvent.O1AXIS));
        System.out.println("Axis Max: " + event.getAxisMax(ScrollEvent.O1AXIS));
}
```

23.7.3 Providing an Interactive Time Axis for Graphs

You can define relative ranges and explicit ranges for the display of time data. You can also add a time selector to allow users to select a time range on the time axis.

23.7.3.1 How to Define a Relative Range of Time Data for Display

You can define a simple relative range of time data to be displayed, such as the last seven days. This will force old data to scroll off the left edge of the graph as new data points are added to the display of an active data graph. Relative time range specifications are not limited to use in active data graphs.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph that displays an axis based on time values on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify a relative range of time data for display:

1. In the Structure window, right click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and specify values for the following attributes:
   a. **TimeRangeMode**: Specify the value `TRM_RELATIVE_LAST` or `TRM_RELATIVE_FIRST` depending on whether the relative range applies to the end of the time range (such as the last seven days) or to the beginning of the time range (such as the first seven days).
   b. **TimeRelativeRange**: Specify the relative range in milliseconds.

      For example, if you wish to specify a seven day range, enter the number of days (`7`) multiplied by the number of milliseconds in a day (`86400000`): `604800000`.

23.7.3.2 How to Define an Explicit Range of Time Data for Display

You can define an explicit range of time data to be displayed, such as the period between March 15 and March 25. In this example, the year, hour, minute, and second use default values because they were not stated in the start and end values.

You store the methods for specifying the start and end dates of the explicit range in the graph's managed bean.

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You will need to complete these tasks:

- Create methods in a managed bean to return the start and end dates for the time range. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

  Example 23–12 shows two sample methods that return the start and end dates for the time range.

Example 23–12 Sample Code to Set Start and End Dates for Explicit Time Range

```
// Add the following imports to your bean
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;

// Add the following variables to your bean
private static Calendar cal1 = new GregorianCalendar (2011,0,2);
private static Calendar cal2 = new GregorianCalendar (2011,0,4);
private static Date m_startDate = cal1.getTime();
private static Date m_endDate = cal2.getTime();

// Add the following methods to your bean
public Date getStartDate(){
  return m_startDate;
}
public Date getEndDate(){
  return m_endDate;
}
```

- Create a graph that displays an axis based on time values on your page. For additional information, see Section 23.2.3, "How to Add a Graph to a Page."

To specify an explicit range of time data for display:

1. In the Structure window, right click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section and specify the values for the following attributes:

   a. **TimeRangeMode**: Choose `TRM_EXPLICIT` from the attribute's dropdown menu.

   b. **TimeRangeStart**: Specify a reference to a method that returns the initial date for the time range.

      For example, for a managed bean named `timeAxisSample` and the `getStartDate()` method referenced in Example 23–12, enter the following for the initial date: `#{timeAxisSample.startDate}`.

   c. **TimeRangeEnd**: Specify a reference to a method that returns the ending date for the time range.

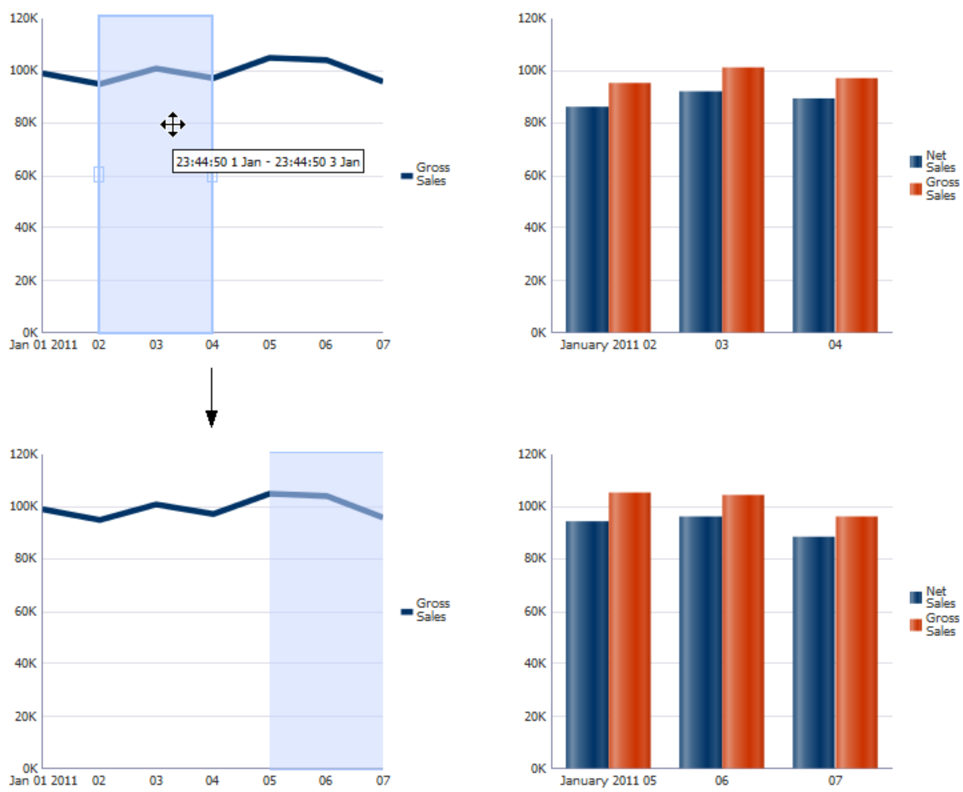
23.7.3.3 How to Add a Time Selector to a Graph

You can add a time selector to any graph that is configured to display a time axis. The time selector permits the user to select a range of data on the time axis. Typically, the time selector is used in master-detail graphs where the detail is based on the time selector's date range.

To add a time selector to a graph, add the `dvt:timeSelector` component to your graph and add methods to a managed or backing bean to return the start and end dates for the range. If you are configuring master-detail graphs, add a listener to the time selector to update the detail graph when the user moves the time selector.

Figure 23–46 shows a simple example of a master-detail graph configured to use a time selector. The bar graph display updates automatically when the user moves the time selector to another date range on the master graph.

Figure 23–46 Time Selector in Master-Detail Graph



Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You will need to complete these tasks:

- Create a graph that displays an axis based on time values on your page. For additional information, see Section 23.2.3, "How to Add a Graph to a Page."

  Area, bar, line, combination, and stock graphs display a time axis when dates of object type `java.util.Date` are specified for the column labels. To use the time selector, ensure that the dates are sorted in ascending order and use regular intervals such as days, weeks, or months.

  For example, the line graph in Figure 23–46 uses sales dates for the o1-axis and gross sales for the series. Figure 23–47 shows the sample data.

Figure 23–47 Sample Data for Time Selector Example

GrossSales	SalesDate	NetSales
99000	2011-01-01	88000
95000	2011-01-02	86000
101000	2011-01-03	92000
97000	2011-01-04	89000
105000	2011-01-05	94000
104000	2011-01-06	96000
96000	2011-01-07	88000

- If you are configuring master-detail graphs, create the graph that will display the detail based on the start and end dates of the time selector.

  For example, the bar graph in Figure 23–46 also uses sales dates for the o1-axis but includes the net sales data in addition to the gross sales.

- Create methods in a managed or backing bean to return the start and end dates for the time range. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

  Example 23–13 shows two sample methods that return the start and end dates for the time selector's time range.

Example 23–13 Sample Methods to Return Start and End Dates for Time Selector

```
// Include these imports in your bean
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

// Add these variables to your bean
private static Calendar cal1 = new GregorianCalendar (2011,0,2);
private static Calendar cal2 = new GregorianCalendar (2011,0,4);
private static Date m_startDate = cal1.getTime();
private static Date m_endDate = cal2.getTime();

// Add these methods to your bean
public Date getTimeAxisStartDate() {
  return m_startDate;
}
public Date getTimeAxisEndDate() {
  return m_endDate;
}
```

- Optionally, add a method to the managed bean for the time selector's listener.

  Example 23–14 shows a sample listener for the time selector displayed in Figure 23–46.

Example 23–14 Sample Code for Time Selector Listener

```
// Include these imports in your bean
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;
import oracle.adf.view.faces.bi.event.TimeSelectorEvent;
import javax.faces.event.AbortProcessingException;

// Add this method to your bean
public void processTimeSelectorEvent(TimeSelectorEvent event) throws
AbortProcessingException
{
```

```
    Date startDate = new Date(event.getStartDate().getTime());
    Date endDate = new Date(event.getEndDate().getTime());
    if (graph2 != null)
    {
      graph2.setTimeRangeStart(startDate);
      graph2.setTimeRangeEnd(endDate);
    }
}
```

In this example, the `graph2` variable is declared as a `UIGraph` in the page's backing bean. When the user changes the time selector range on the master graph, the listener code sets the new time range on the detail graph.

To add a time selector to a graph:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType*** > **ADF Data Visualizations** > **Time Selector**.
2. Right-click the **dvt:timeSelector** node and choose **Go to Properties**.
3. In the Properties window, enter values for the following attributes:
   - **ExplicitStart**: Specify a reference to a method that returns the initial starting date for the time range.

     For example, for a bean named `timeSelectorDemo` and the `getTimeAxisStartDate()` method referenced in Example 23–13, enter the following for the initial date: `#{timeSelectorDemo.timeAxisStartDate}`.
   - **ExplicitEnd**: Specify a reference to a method that returns the initial ending date for the time range.
   - **Mode**: From the attribute's dropdown list, select `EXPLICIT` to enable the time selector display. By default, this attribute is set to `OFF`.
   - **FillColor**: From the attribute's dropdown list, select the color for the fill of the time selector.
   - To display the data points behind the time selector, use the **Transparency** slider to select a value below 100%.

     For example, set the **Transparency** slider to 53% to duplicate the time selector in Figure 23–46. This has the effect of changing the fill color value from a 6 digit hexadecimal value to an 8 digit value. The first two digits represent the transparency for the fill color. By default, the fill color is set to `#000000`, and the time selector's fill color is opaque.
   - Optionally, to enable transparency, from the **FillTransparent** dropdown list, select `true`.

     If you set **FillTransparent** and **BorderTransparent** to `true`, the time selector will not be displayed, but the user can still select it.
   - **BorderColor**: From the attribute's dropdown list, select the color for the border of the time selector.
   - Optionally, to enable transparency, from the **BorderTransparent** dropdown list, select `true`.
   - Optionally, in the **TimeSelectorListener** field, specify a reference to a method that returns the listener for the time selector.

     For example, for a managed bean named `timeSelectorDemo` and the `processTimeSelectorEvent` method referenced in Example 23–14, enter the

following for the time selector listener:
`#{timeSelectorDemo.processTimeSelectorEvent}`.

Example 23–15 shows the code on the JSF page for the example time selector.

Example 23–15 Time Selector Code on JSF Page

```
<dvt:timeSelector explicitStart="#{timeSelectorDemo.timeAxisStartDate}"
                  explicitEnd="#{timeSelectorDemo.timeAxisEndDate}"
                  fillColor="#88c6d6ff" borderColor="#a5c6ff" mode="EXPLICIT"
             timeSelectorListener="#{timeSelectorDemo.processTimeSelectorEvent}"/>
```

4. If you created a detail graph, update the detail graph to use an explicit time range and configure it to update when the time selector changes.
    1. In the Structure window, right-click the detail graph node and choose **Go to Properties**.
    2. In the Properties window, expand the **Appearance** section.
    3. From the **TimeRangeMode** attribute's dropdown list, select `TRM_EXPLICIT`.
    4. In the **TimeRangeStart** field, specify a reference to a method that returns the starting time for the time range.

       For example, for a bean named `timeSelectorDemo` and the `getTimeAxisStartDate()` method referenced in Example 23–13, enter the following for the initial date: `#{timeSelectorDemo.timeAxisStartDate}`.
    5. In the **TimeRangeEnd** field, specify a reference to a method that returns the ending time for the time range.
    6. In the Properties window, expand the **Behavior** section.
    7. In the **PartialTriggers** field, enter the ID of the master graph to enable the detail graph to update when the user changes the time selector range.

       For example, enter `::graph1` to reference the ID of the line graph in Figure 23–46. You can also choose **Edit** from the **PartialTriggers** dropdown menu to select the partial trigger.

23.7.4 Adding Alerts and Annotations to Graphs

Alerts define a data value on a graph that must be highlighted with a separate symbol, such as an error or warning. An icon marks the location of the alert. When the cursor moves over the alert icon, the text of that alert is displayed.

Annotations are associated with a data value on a graph to provide information when the cursor moves over the data value.

23.7.4.1 How to Add Alerts to Graphs

An unlimited number of alerts can be defined for a graph using `dvt:alert` tags. The alerts are wrapped in a `dvt:alertSet` tag which is a child of the graph tag. For each alert, you must specify the image source (`imageSource`) and location that you want the alert to display on the x-axis or ordinal axis (`xValue`) and on the y-axis (`yValue`). If your graph uses a y2-axis, you can use the `YValueAssignment` attribute to associate the `yValue` with the y2-axis. You can also specify the text to display when the user hovers the mouse over the alert.

Example 23–16 shows the code for an alert displayed on a line graph. In this example, the `xValue` and `yValue` attributes are defined in the `alertBean` managed bean. At runtime, the user can modify the alert's location on both axes by changing the spin box or date selector.

Example 23–16 Sample Code for Graph Alerts

```
<af:panelGroupLayout>
  <af:outputText value="Use spin box and date selector to change alert location."
                 id="ot2"/>
  <af:panelGroupLayout id="pgl2" layout="horizontal">
    <dvt:lineGraph id="graph1"
                   subType="LINE_VERT_ABS"
                   tabularData="#{alertBean.lineList}"
                   timeAxisType="TAT_MIXED_FREQUENCY_STRICT"
                   partialTriggers="::al1 ::yValue"
                   shortDesc="Line Graph">
      <dvt:graphTitle text="Line Graph"/>
      <dvt:alertSet>
        <dvt:alert xValue="#{alertBean.alertDate}"
                   yValue="#{alertBean.alertYValue}"
                   imageSource="#{resource['images:alert_icon.png']}"
                   text="Alert Example"/>
      </dvt:alertSet>
      <dvt:graphFootnote text="Alerts assigned to location and date"/>
    </dvt:lineGraph>
    <af:panelGroupLayout layout="vertical" id="pgl8">
      <af:inputDate label="Date" id="al1" value="#{alertBean.alertDate}"
                    autoSubmit="true"/>
      <af:inputNumberSpinbox label="Y Value" id="yValue"
                             value="#{alertBean.alertYValue}"
                             autoSubmit="true" stepSize="10"/>
    </af:panelGroupLayout>
</af:panelGroupLayout>
```

You specify values for the alert’s location attributes through the Properties window. The `xValue` attribute can be a string, double, or date. The `yValue` must be defined as a double.

If your `xValue` is a date, you must define a method that sets the alert’s `xValue` in the graph’s managed bean. Example 23–17 shows code that will create a sample array of tabular data and define the alert’s methods to change the alert date and value to match the user’s selection. For additional information about tabular data, see Section 23.2.5, "How to Create a Graph Using Tabular Data." For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

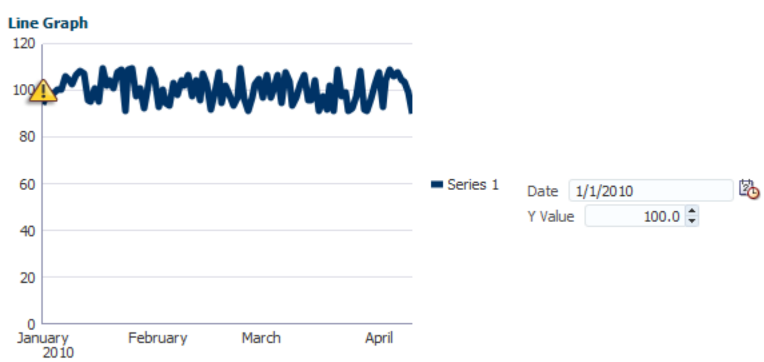
Example 23–17 Sample Managed Bean for Graph Alerts

```
import java.util.Date;
import java.util.ArrayList;
import java.util.Random;
import java.util.List;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class alertBean {
  private List m_lineList;
  private Double m_alertYValue;
  private static Calendar cal1 = new GregorianCalendar (2010,0,1);
  private static Calendar cal2 = new GregorianCalendar (2010,0,1);
  private Date m_date;
  public alertBean(){
```

```
        m_alertYValue = new Double(100);
        m_date = cal1.getTime();
        Random random = new Random();
          m_lineList = new ArrayList();
          Date newDate = cal2.getTime();
          for (int i = 0; i < 100; i ++){
             double number = random.nextDouble() * 20+90;
             m_lineList.add(new Object[]{"Group "+i,"Series 1", newDate});
             m_lineList.add(new Object[]{"Group' "+i,"Series 1",
                           new Double(number)});
             newDate = new Date(newDate.getTime() + 86400000);
          }
      }
        public List getLineList(){
            return m_lineList;
        }
        public Date getAlertDate(){
            return m_date;
        }
        public void setAlertDate(Date date){
            m_date = date;
        }
        public Double getAlertYValue(){
            return m_alertYValue;
        }
        public void setAlertYValue(Double value){
            m_alertYValue = value;
        }
}
```

Figure 23–48 shows the initial page that is rendered when the application starts. If you change the date or enter a different value for the y-axis, the alert icon will move to the new location.

Figure 23–48 Line Graph with Customizable Alert Location



Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child tags can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You should already have a graph on your page. If you do not, follow the instructions in this chapter to create a graph. For information, see Section 23.2.3, "How to Add a Graph to a Page."

To add an alert to a graph:

1. Optionally, create a managed bean and add the code that will set and get the alert values. For additional help, see Section 3.6, "Creating and Using Managed Beans."

   For example, to duplicate the alert displayed in Figure 23–48, create a managed bean named `popupSample.java` using the code in Example 23–17.

2. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Insert Inside *GraphType* > ADF Data Visualizations > Alert Set**.

3. Right-click the **dvt:alert** node and choose **Go to Properties**.

4. In the Properties window, set the following attributes:

   - **ImageSource**: Specify the path to the alert's icon using the attribute's dropdown menus to select an existing image or choose **Edit** to add a new one to the application.
   - **Text**: Specify optional text that appears in a tooltip when a mouse pointer hovers over the icon.
   - **XValue**: Specify a value for the x-axis or ordinal axis. Optionally, use the attribute's dropdown menu to access the **EL Expression Builder** and select the method that sets the alert's `xValue`.

     For example, for the `alertBean` managed bean in Example 23–17 and the method `setAlertDate`, set the `xValue` to the following value: `#{alertBean.alertDate}`.
   - **YValue**: Specify a value for the y-axis. Optionally, use the attribute's dropdown menu to access the **EL Expression Builder** and select the method that sets the alert's `yValue`.

     For example, for the `alertBean` managed bean in Example 23–17 and the method `setAlertYValue`, set the `yValue` to the following value: `#{alertBean.alertDate}`.
   - **YValueAssignment**: Identify which axis to associate with the `yValue` attribute. Valid values are: `Y1AXIS` (default) or `Y2AXIS` (if configured).

     In the managed bean example in Example 23–17, the `yValue` is associated with the `Y1AXIS` so there is no need to change the default.

5. To add additional alerts, in the Structure window, right-click the **dvt:alertSet** and choose **Insert Inside Alert Set** > **Alert** for each new alert.

6. Repeat Step 3 and Step 4 to configure the new alert.

23.7.4.2 Adding Annotations to Graphs

An unlimited number of annotations can be defined for a graph using `dvt:annotation` tags, and multiple annotations can be associated with a single data value. The annotations are wrapped in a `dvt:annotationSet` tag that is a child of the graph tag.

The data marker associated with the annotation is defined using these attributes of the `dvt:annotation` tag:

- `series`: Specifies the zero-based index of a series in a graph. In most graphs, each series appears as a set of markers that are the same color. For example, in a multiple pie graph, each yellow slice might represent sales of shoes, while each green slice represents the sales of boots. In a bar graph, all of the yellow bars might represent the sales of shoes, and the green bars might represent the sales of boots.

- group: Specifies the zero-based index of a group in a graph. Groups appear differently in different graph types. In a clustered bar graph, each cluster of bars is a group. In a stacked bar graph, each stack is a group. In a multiple pie graph, each pie is a group.

Example 23–18 shows a set of annotations for an area graph.

Example 23–18 Sample Code for a Set of Annotations

```
<dvt:areaGraph>
  <dvt:annotationSet>
    <dvt:annotation series="0" group="0" text="annotation #1"/>
    <dvt:annotation series="0" group="7" fillColor="#55FFFF00"
                    borderColor="#55FF0000" text="second annotation"/>
  </dvt:annotationSet>
</dvt:areaGraph>
```

You can control the position of the annotation in the plot area of a graph using these attributes:

- position: Specifies the type of positioning to use for the annotation. Valid values are:
  - dataValue (default): Positions the annotation by the data value defined in the series and group attributes. Overlap with other annotations is avoided.
  - absolute: Positions the annotation at the exact point defined by the xValue and the yValue in graphs with both those axes. Overlap with other annotations is not avoided.
  - percentage: Positions the annotation at the exact point defined by using the xValue and yValue as a percentage between 0 and 100 of the plot area of graphs with both those axes. Overlap with other annotations is not avoided.
- xValue: Specifies the X value at which to position the annotation. This setting only applies when the annotation position is absolute or percentage.
- yValue: Specifies the Y value at which to position the annotation. This setting only applies when the annotation position is absolute or percentage.
- horizontalAlignment: Specifies the horizontal positioning of the annotation. This setting only applies when the annotation position attribute is absolute or percentage. Valid values are LEFT (default), CENTER, LEADING, or RIGHT.
- verticalAlignment: Specifies the vertical positioning of the annotation. This setting only applies when the annotation position attribute is absolute or percentage. Valid values are CENTER (default), TOP, or BOTTOM.

23.7.5 Creating Drillable Graphs

If your application uses the Fusion technology stack, then you can create a drillable graph. Drillable graphs are bound to an ADF data control and display a detailed view of data displayed by an area, line, or marker. To make a graph drillable, set the drillingEnabled attribute for the <type>Graph component to true. The default value is false.

You must also update the graph's page definition file to define the drilling hierarchy and specify how you want the aggregated data displayed. For more information about adding drilling support to databound graphs, see the "Creating Databound Graphs" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

23.7.6 Adding Drag and Drop to Graphs

The ADF Faces framework provides the ability to drag and drop items from one place to another on a page. Bubble and scatter graphs can be configured as a drag source to allow moving markers from one position to another in the graph plot area, by adding and configuring a child `af:dragSource` component. All data axis graphs, those with x and y positions, can be configured as a drop target. For example, you can insert a bubble graph marker into a bar graph, by adding and configuring a child `af:dropTarget` component.

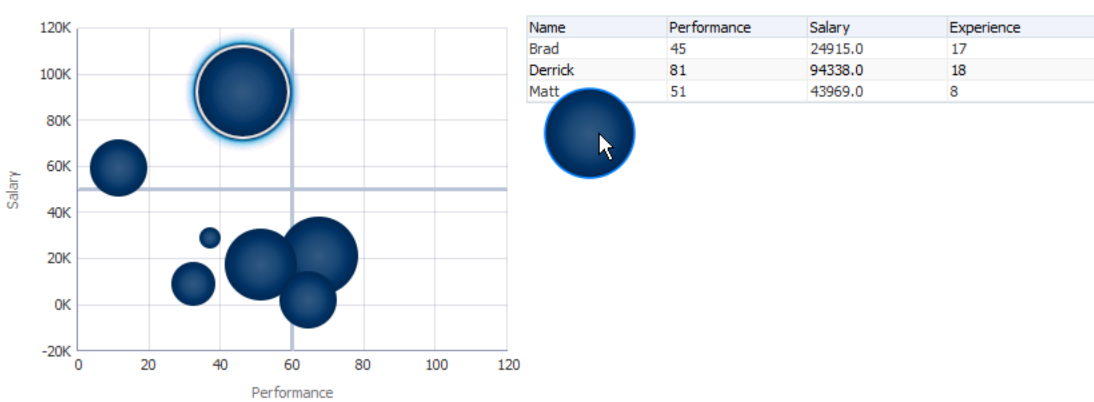
23.7.6.1 How to Add Drag and Drop to Graphs

Graphs can be configured to support these operations:

- Drag between graphs
- Drag from a graph into another ADF component
- Drag a scatter/bubble marker within the plot area of a graph
- Drag a scatter/bubble marker to another component
- Drag multiple markers

For example, you may want to drag the data markers in a bubble graph to display their values in a table view, or conversely, you may wish to drag data values from a table for display in a bubble graph. Figure 23–49 shows a bubble graph configured as a drag source and a drop target to accomplish this functionality.

Figure 23–49 Bubble Graph as Drag Source and Drop Target



You define the methods that respond to the drag and drop events in a managed bean. Example 23–19 shows a code sample for adding the drag and drop listeners used in Figure 23–49. For information about using managed beans, see Section 3.6, "Creating and Using Managed Beans."

Example 23–19 Managed Bean Sample for Handling Drag and Drop Targets

```
public class dragAndDrop {
  public DnDAction fromGraphDropListener(DropEvent event) {
    // Get the ComponentHandle from the transferable
    Transferable transferable = event.getTransferable();
```

```
    GraphSelectionSet selectionSet =
transferable.getData(GraphSelectionSet.class);
    // Now change each marker based on the DropEvent's proposed action
    DnDAction proposedAction = event.getProposedAction();
   for(GraphSelection selection : selectionSet) {
      Employee emp = findEmployee(m_graphList, selection);
      if(emp == null)
        return DnDAction.NONE;
      if(proposedAction == DnDAction.COPY) {
        m_tableModel.add(new Employee("Copy of " + emp.getName(), emp));
      }
      else if(proposedAction == DnDAction.LINK) {
        m_tableModel.add(new Employee("Link to " + emp.getName(), emp));
      }
      else if(proposedAction == DnDAction.MOVE) {
        m_graphList.remove(emp);
        m_tableModel.add(emp);
      }
    }
 RequestContext.getCurrentInstance().addPartialTarget(event.getDragComponent());
        return proposedAction;
  }
  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();
  }
```

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child components can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You will need to complete these tasks:

- Create a bubble graph on your page. For more information, see Section 23.2.3, "How to Add a Graph to a Page."
- Create and configure the table component as a drag source and drop target. Example 23–20 shows a code sample for creating a table. For information about creating tables, see Chapter 12.3, "Displaying Data in Tables." For additional detail about creating drag source and drop targets, see Chapter 36, "Adding Drag and Drop Functionality."

Example 23–20 Code Sample for Table Drag Source and Drop Target

```
<af:table id="table" value="#{dragAndDrop.tableModel}"
                     var="row" width="420" inlineStyle=" height:300px;"
                     columnStretching="last">
  <af:dragSource actions="COPY MOVE LINK" defaultAction="MOVE"
                 discriminant="fromTable"/>
  <af:dropTarget dropListener="#{dragAndDrop.fromGraphDropListener}"
                 actions="COPY MOVE LINK">
    <af:dataFlavor
          flavorClass="oracle.adf.view.faces.bi.component.graph.GraphSelectionSet"
          discriminant="fromGraph"/>
  </af:dropTarget>
  <af:column headerText="Name" id="c1">
    <af:outputText value="#{row.name}" id="ot5"/>
  </af:column>
  <af:column headerText="Performance" id="c2">
    <af:outputText value="#{row.performance}" id="ot6"/>
  </af:column>
  <af:column headerText="Salary" id="c3">
    <af:outputText value="#{row.salary}" id="ot7"/>
  </af:column>
  <af:column headerText="Experience" id="c4">
    <af:outputText value="#{row.experience}" id="ot8"/>
  </af:column>
</af:table>
```

- Create the methods to respond to the drag and drop events in a managed bean. Example 23–19 shows a code sample for handling the drag and drop events. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

To add drag and drop to a bubble graph:

1. In the Structure window, right-click the **dvt:bubbleGraph** component and choose **Insert Inside Bubble > ADF Faces > Drag Source**.
2. In the Properties window, set the following attributes:
   - **Actions**: Specify the drag and drop actions supported by this drag source. The actions must be a space-delimited list with all caps in any particular order for the available values of `COPY`, `LINK`, or `MOVE`. The default value is `COPY`.
   - **DefaultAction**: Specify the default drag and drop action supported by this drag source. Possible actions are `COPY`, `LINK`, or `MOVE`.
   - **Discriminant**: Specify the discriminant for the default data flavors generated by this drag source. The discriminant is used to segregate drags from this drag source. Please note that drag and drop can only be performed between compatible drag sources and drop targets. The discriminant is used for the compatibility purpose. The discriminants of the data flavors generated by the

default drag source must match the allowed discriminants on the allowed data flavors of the drop target.

3. In the Structure window, right-click the **dvt:bubbleGraph** node and choose **Insert Inside Bubble** > **ADF Faces** > **Drop Target**.

4. 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.

   Example 23–21 shows a code sample for adding drag and drop operations to the bubble graph in Figure 23–49. To specify the **DropListener** used in this example, enter: `#{dragAndDrop.fromTableDropListener}`.

Example 23–21 Code Sample for Bubble Graph Drag and Drop

```
<dvt:bubbleGraph shortDesc="Graph" dataSelection="multiple"
                 markerTooltipTemplate="GROUP_LABEL NEW_LINEPerformance: X_VALUE
                 NEW_LINESalary: Y_VALUE NEW_LINEExperience: Z_VALUE"
                 value="#{dragAndDrop.graphModel}" id="bg1">
  <dvt:x1Title text="Performance"/>
  <dvt:y1Title text="Salary"/>
  <dvt:x1Axis axisMaxValue="120" axisMaxAutoScaled="false"/>
  <dvt:y1Axis axisMaxValue="120000" axisMaxAutoScaled="false"/>
  <dvt:legendArea rendered="false"/>
  <af:dragSource actions="COPY MOVE LINK" defaultAction="MOVE"
                 discriminant="fromGraph"/>
  <af:dropTarget actions="COPY MOVE LINK"
                 dropListener="#{dragAndDrop.fromTableDropListener}">
    <af:dataFlavor flavorClass="org.apache.myfaces.trinidad.model.RowKeySet"
                   discriminant="fromTable"/>
  </af:dropTarget>
</dvt:bubbleGraph>
```

5. 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** used in Example 23–21, enter: `org.apache.myfaces.trinidad.model.RowKeySet`.

23.7.6.2 What You May Need to Know About Adding Drag and Drop to Graphs

The graph and table data in Figure 23–49 is specified in the `value` attribute of the graph and table. In this example, a managed bean includes code to set up a simple `Employee` class. For more information about providing data to the graph and table, see Section 22.3, "Providing Data for ADF Data Visualization Components."

See Section F.2 for the complete version of the managed bean that creates the `Employee` class and sample methods used in the example.

For additional information about configuring drag and drop in graphs, see Chapter 36, "Adding Drag and Drop Functionality."

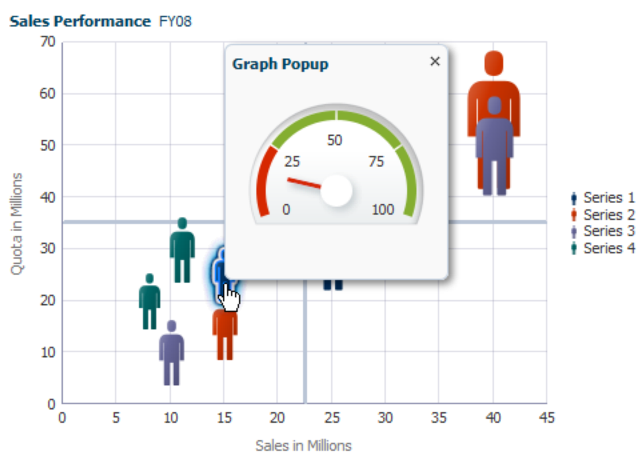
23.7.7 How to Add Popups to Graphs

Graph child component `seriesSet` 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 Chapter 15, "Using Popup Dialogs, Menus, and Windows."

For example, you may want to associate a popup displaying information in a note window with the data markers in a scatter graph series. Figure 23–50 shows a scatter graph with a data marker clicked to display a gauge in a note window with data about a specific marker.

Figure 23–50 Scatter Graph Data Marker Popup Note Window



Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child components can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You will need to complete these tasks:

- Add a graph to your page. For more information, see Section 23.2.3, "How to Add a Graph to a Page."

  To duplicate the gauge popup example in Figure 23–50, create a scatter graph.

- Create the popup components for data points in a graph series to reference. Figure 23–22 shows a code sample for the popup to be referenced when the user clicks on a data point in the graph series displayed in Figure 23–50.

Example 23–22 Code Sample for the Graph Data Marker Popup Window

```
<af:popup id="graphPopup" launcherVar="source"
          eventContext="launcher"
          clientComponent="true" contentDelivery="lazyUncached">
  <af:setPropertyListener from="#{source.seriesKey}"
                          to="#{popupSample.seriesKey}" type="popupFetch"/>
  <af:setPropertyListener from="#{source.groupKeys}"
```

```
                          to="#{popupSample.groupKeys}" type="popupFetch"/>
  <af:panelWindow title="Graph Popup" id="pw1">
    <dvt:gauge shortDesc="Gauge" value="#{popupSample.gaugeValue}"
               gaugeType="DIAL" animationOnDisplay="auto"
               inlineStyle="width:150px;height:120px;" minValue="0"
               maxValue="100" contentDelivery="immediate" id="g3">
      <dvt:bottomLabel text="Sales" position="LP_NONE"/>
      <dvt:metricLabel position="LP_NONE"/>
      <dvt:thresholdSet>
        <dvt:threshold thresholdMaxValue="#{popupSample.quotaValue}"
                       text="Under Quota"/>
        <dvt:threshold fillColor="#84AE31" text="Above Quota"/>
      </dvt:thresholdSet>
      <dvt:indicator useThresholdFillColor="true"/>
      <dvt:tickLabel content="TC_MIN_MAX TC_INCREMENTS"/>
      <dvt:tickMark content="TC_NONE" majorTickCount="5"/>
    </dvt:gauge>
  </af:panelWindow>
</af:popup>
```

- Add methods to a managed bean to define the listener methods that will respond to the popup request. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

  Example 23–23 shows the sample code used to respond to the popup request in the gauge popup example. In this example, the scatter graph is using default data values. For more information about providing data to graphs, see Section 22.3, "Providing Data for ADF Data Visualization Components."

Example 23–23 Managed Bean Sample Code for the Graph Data Marker Popup

```
import java.util.List;
import java.util.Set;
import oracle.adf.view.faces.bi.component.gauge.UIGauge;
import oracle.adf.view.faces.bi.component.graph.UIGraph;
import oracle.adf.view.faces.bi.model.KeyMap;
import oracle.dss.graph.DataType;
public class popupSample {
  private UIGraph m_graph;
  private UIGauge m_gauge;
  private KeyMap m_seriesKey;
  private List<KeyMap> m_groupKeys;
  public UIGraph getGraph() {
      if(m_graph == null)
         m_graph = new UIGraph();
      return m_graph;
  }
  public void setGraph(UIGraph graph) { m_graph = graph; }
  public UIGauge getGauge() {
      if(m_gauge == null)
         m_gauge = new UIGauge();
      return m_gauge;
  }
  public void setGauge(UIGauge gauge) { m_gauge = gauge;}
  public KeyMap getSeriesKey() { return m_seriesKey; }

  public void setSeriesKey(KeyMap key) { m_seriesKey = key; }
  public List<KeyMap> getGroupKeys() { return m_groupKeys; }
  public void setGroupKeys(List<KeyMap> keys) { m_groupKeys = keys; }
  private int m_gaugeValue = 0;
```

```
  public int getGaugeValue() {
    // Reestablish context
    m_graph.setDataKey(m_seriesKey, m_groupKeys);
    // Fetch the x value
    Object val = m_graph.getDataValue(DataType.X_VALUE);
    if(val instanceof Number) {
      Number number = (Number) val;
      m_gaugeValue = (int) number.doubleValue();
    }
    return m_gaugeValue;
  }
  private int m_quotaValue = 0;
  public int getQuotaValue() {
    // Reestablish context
    m_graph.setDataKey(m_seriesKey, m_groupKeys);
    // Fetch the y value
    Object val = m_graph.getDataValue(DataType.Y_VALUE);
    if(val instanceof Number) {
      Number number = (Number) val;
      m_quotaValue = (int) number.doubleValue();
    }
    return m_quotaValue;
  }
}
```

To add a popup to a graph series set:

1. In the Structure window, right-click the graph child **dvt:seriesSet** component and choose **Insert Inside Series Set** > **Show Popup Behavior**.
2. Right-click the **af:showPopupBehavior** node and choose **Go to Properties**.
3. In the Properties window, set the following attributes:
   - **PopupId**: Enter the ID of the popup, relative to the containing component. An ID beginning with a colon will be treated as absolute after trimming off the colon.

     For example, to reference the gauge popup in Example 23–22, enter: `::graphPopup`.
   - **TriggerType**: Enter the event type that will trigger the popup being displayed. Valid values for graph `seriesSet` components are `action`, `click`, and `mouseHover`.

     For example, to continue with the gauge popup example, enter: `click`.
   - **Align**: From the dropdown list, choose how the popup should be aligned with the `seriesSet` component.

     This attribute does not have a default setting and is normally required. However, the attribute does not require setting if the popup type is `af:panelWindow` or `af:dialog`. These types of popups can be manually repositioned using drag-and-drop and will open by default at the center of the browser window.

     To continue with the gauge popup example, leave this field blank since the gauge popup example is using an `af:panelWindow` type.
   - **AlignID**: Enter the ID of the component relative to which the popup will be aligned. An ID beginning with a colon will be treated as absolute after trimming off the colon.

This attribute is also normally required unless the popup type is `af:panelWindow` or `af:dialog`. To continue with the gauge popup example, leave this field blank.

Example 23–24 shows the code sample for associating a popup component with a graph series set component.

Example 23–24 Code Sample for Popup Associated With Series Set Component

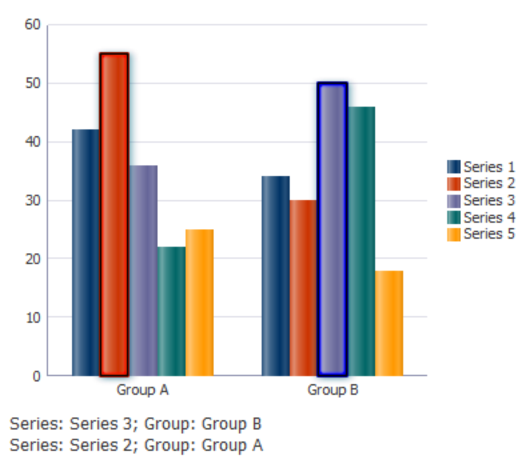
```
<dvt:scatterGraph dataSelection="single"
                  shortDesc="Scatter Graph with Click Popup"
                  binding="#{popupSample.graph}"
                  inlineStyle="width:500px;height:350px;" id="g2">
  <dvt:seriesSet defaultMarkerShape="MS_HUMAN">
    <af:showPopupBehavior triggerType="click" popupId="::graphPopup"/>
  </dvt:seriesSet>
  <dvt:x1Title text="Sales in Millions"/>
  <dvt:y1Title text="Quota in Millions"/>
  <dvt:graphTitle text="Sales Performance"/>
  <dvt:graphSubtitle text="FY08"/>
</dvt:scatterGraph>
```

23.7.8 Adding Data Marker Selection Support for Graphs

Add selection support to respond programmatically when a user selects one or more of the graph's data markers.

For example, Figure 23–51 displays a bar graph supporting single and multiple selection to output information about each selected series. To make multiple selections, users press **Control** on the keyboard while selecting the data markers.

Figure 23–51 Bar Graph Displaying Multiple Selection Support



23.7.8.1 How to Add Selection Support to Graphs

To add selection support, create a listener in a managed bean that will handle the `SelectionEvent` and perform the needed logic. You then enable selection support in the graph's `dataSelection` attribute and bind the `selectionListener` attribute of the graph to that listener.

Example 23–25 shows sample code to create a managed bean that returns the selection state as the formatted string displayed below the bar graph in Figure 23–51.

Example 23–25 Managed Bean Example for Graph Selection Support

```
import java.util.List;
import java.util.Set;
import oracle.adf.view.faces.bi.component.graph.DataSelection;
import oracle.adf.view.faces.bi.component.graph.GraphSelection;
import oracle.adf.view.faces.bi.event.graph.SelectionEvent;
import oracle.adf.view.faces.bi.model.KeyMap;

public class graphSelection {
  public void selectionListener(SelectionEvent selectionEvent) {
    StringBuilder eventInfo = new StringBuilder();
    Set<? extends GraphSelection> selectionSet =
selectionEvent.getGraphSelection();
    eventInfo.append(convertSelectionStateToString(selectionSet));

    // Store on the selection string
    m_selectionInfo = eventInfo.toString();
  }

  /**
   * Returns the selection state as a formatted String with one selected data
point per line.
   * @param selectionSet
   * @return
   */
  public static String convertSelectionStateToString(Set<? extends GraphSelection>
selectionSet) {
    StringBuilder selectionState = new StringBuilder();
    for(GraphSelection selection: selectionSet) {
      if(selection instanceof DataSelection) {
        DataSelection ds = (DataSelection) selection;
        Set seriesKeySet = ds.getSeriesKey().keySet();
        for(Object key : seriesKeySet) {
          selectionState.append(key).append(":
").append(ds.getSeriesKey().get((String)key));
        }

        List<KeyMap> groupKeys = ds.getGroupKeys();
        for(KeyMap groupKey : groupKeys) {
          Set groupKeySet = groupKey.keySet();
          for(Object key : groupKeySet) {
            selectionState.append("; ").append(key).append(":
").append(groupKey.get((String)key));
          }
        }

        selectionState.append("<br>");
      }
    }

    return selectionState.toString();
  }

  private String m_selectionInfo = "Select a marker to see information here.
Multiple objects can be selected by holding CTRL while selecting.";
  public String getSelectionInfo() {
    return m_selectionInfo;
  }
}
```

Example 23–26 shows the code sample for configuring the JDeveloper page for multiple selection support and to bind the `selectionListener` attribute of the graph to the selection listener. The sample uses the `af:outputFormatted` component to display the selected information on the page.

Example 23–26 Code Sample for Configuring Graph Selection Support on a Page

```
<af:panelGroupLayout id="pgl1">
  <dvt:barGraph id="graph1" subType="BAR_VERT_CLUST" shortDesc="BarGraph"
                selectionListener="#{graphSelection.selectionListener}"
                dataSelection="multiple">
    <dvt:background>
      <dvt:specialEffects/>
    </dvt:background>
    <dvt:graphPlotArea/>
    <dvt:seriesSet>
      <dvt:series/>
    </dvt:seriesSet>
    <dvt:o1Axis/>
    <dvt:y1Axis/>
    <dvt:legendArea automaticPlacement="AP_NEVER"/>
  </dvt:barGraph>
  <af:outputFormatted id="selectionText"
                      inlineStyle="font-size:120.0%;"
                      partialTriggers="graph1"
                      value="#{graphSelection.selectionInfo}"/>
</af:panelGroupLayout>
```

Before you begin:

It may be helpful to have an understanding of how graph attributes and graph child components can affect functionality. For more information, see Section 23.2.2, "Configuring Graphs."

It may also be helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 23.1.3, "Additional Functionality for Graph Components."

You may also find it helpful to understand event handling in ADF Faces. For information, see Chapter 6, "Handling Events."

You will need to complete these tasks:

- Add a graph to your page. For more information, see Section 23.2.3, "How to Add a Graph to a Page."

  To duplicate the multiple selection support example in this section, create a bar graph.

- Add methods to a managed bean to define the listener methods that will respond to the selection events. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

To add selection support to graphs:

1. In the Structure window, right-click the **dvt:*typeGraph*** node and choose **Go to Properties**.
2. In the Properties window, expand the **Behavior** section and specify the values for the following attributes:

   a. **DataSelection**: Specify `single` or `multiple` to enable selection support for single or multiple data markers. The default is none which means that selection is not enabled by default.

   b. **SelectionListener**: Specify a reference to the selection listener.

      For example, to specify the `selectionListener` method in a managed bean named `graphSelection.java`, enter the following in the SelectionListener field: `#{graphSelection.selectionListener}`.

3. Complete any additional configuration as needed.

   For example, to duplicate the multiple selection example in this section, add the following code to your page:

```
<af:outputFormatted id="selectionText"
    inlineStyle="font-size:120.0%;"
    partialTriggers="graph1" value="#{graphSelection.selectionInfo}"/>
```

23.7.8.2 What You May Need to Know About Graph Data Marker Selection

The selection listener responds to click events on graph data markers only.

JDeveloper also provides a `clickListener` listener that can respond to click events on other graph components. The click listener, however, provides only single selection support and does not provide the same hover and click feedback that the `selectionListener` listener can provide. The `clickListener` attribute is also not available on newer components, and its use is generally discouraged in favor of the selection listener.

23.7.9 Configuring Graph Context Menus

Context menus can be defined for graph components using these context menu facets:

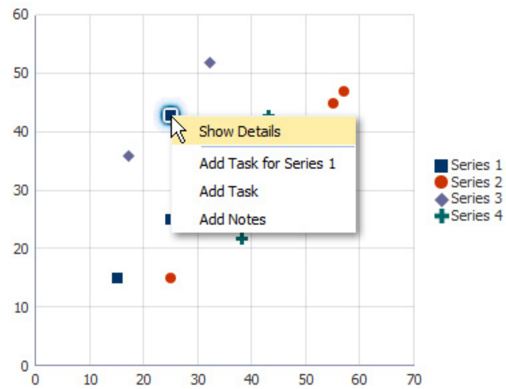
- `bodyContextMenu`: Specifies a context menu that is displayed on non selectable elements in the graph component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the graph component.
- `multiSelectContextMenu`: Specifies a content menu that is displayed when multiple elements are selected in the graph 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 and supported for the specific graph type. Context menus are currently only supported in Flash.

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. For more information, see Section 22.2.4, "Context Menus for Graphs, Gauges, Treemaps, and Sunbursts."

For example, Figure 23–52 shows a scatter graph context menu with custom menu items.

Figure 23–52 Scatter Graph Custom Context Menu



Example 23–27 shows a code sample for configuring a scatter graph context menu.

Example 23–27 Code Sample for Scatter Graph Context Menu

```
<dvt:scatterGraph binding="#{contextMenu.graph}" subType="SCATTER"
                  dataSelection="multiple" id="graph" shortDesc="ScatterGraph">
  <f:facet name="contextMenu">
    <af:popup contentDelivery="lazyUncached" id="p1">
      <af:menu id="m1">
        <af:commandMenuItem text="Show Details"
                            actionListener="#{contextMenu.menuItemListener}"
                            id="cmi1"/>
        <af:group id="g1">
          <af:commandMenuItem text="Add Task for #{contextMenu.currentSeriesId}"
                              actionListener="#{contextMenu.menuItemListener}"
                              id="cmi2"/>
          <af:commandMenuItem text="Add Task"
                              actionListener="#{contextMenu.menuItemListener}"
                              id="cmi3"/>
          <af:commandMenuItem text="Add Notes"
                              actionListener="#{contextMenu.menuItemListener}"
                              id="cmi4"/>
        </af:group>
      </af:menu>
    </af:popup>
  </f:facet>
  <f:facet name="bodyContextMenu">
    <af:popup contentDelivery="immediate" id="p2">
      <af:menu id="m2">
        <af:goMenuItem text="www.oracle.com"
                       destination="http://www.oracle.com"
                       id="gmi1"/>
      </af:menu>
    </af:popup>
  </f:facet>
  <f:facet name="multiSelectContextMenu">
    <af:popup contentDelivery="lazyUncached" id="p3">
      <af:menu id="m3">
        <af:commandMenuItem text="Compare Selected Objects"
                            actionListener="#{contextMenu.menuItemListener}"
                            id="cmi5"/>
      </af:menu>
    </af:popup>
  </f:facet>
```

```
</dvt:scatterGraph>
```

Example 23–28 shows a code sample for a managed bean to create a custom context menu. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

Example 23–28 Managed Bean to Create Custom Context Menu

```
import java.util.Set;
import javax.faces.component.UIComponent;
import javax.faces.event.ActionEvent;
import oracle.adf.view.faces.bi.component.graph.DataSelection;
import oracle.adf.view.faces.bi.component.graph.GraphSelection;
import oracle.adf.view.faces.bi.component.graph.UIGraph;
import oracle.adf.view.faces.bi.model.KeyMap;
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 ContextMenu {
  private RichOutputFormatted m_outputFormatted;
  public RichOutputFormatted getOutputFormatted() {
    if(m_outputFormatted == null)
      m_outputFormatted = new RichOutputFormatted();
    return m_outputFormatted;
  }
  public void setOutputFormatted(RichOutputFormatted text) {
    m_outputFormatted = text;
  }
  private String m_status = "Click Menu Item for Status";
  public String getStatus() {
    return m_status;
  }
  private UIGraph m_graph;
  public UIGraph getGraph() {
    if(m_graph == null)
      m_graph = new UIGraph();
    return m_graph;
  }
  public void setGraph(UIGraph graph) {
    m_graph = graph;
  }
  public String getCurrentSeriesId() {
    if(m_graph != null) {
      Set<? extends GraphSelection> set = m_graph.getSelection();
      if(set != null && !set.isEmpty()) {
        GraphSelection selection = set.iterator().next();
        if(selection instanceof DataSelection) {
          DataSelection dataSelection = (DataSelection) selection;
          KeyMap seriesKey = dataSelection.getSeriesKey();
          Set seriesKeySet = seriesKey.keySet();
          for(Object key : seriesKeySet) {
            return seriesKey.get((String)key);
          }
        }
      }
    }
    return null;
  }
  /**
```

```
   * 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>");
      // If graph data is selected, add that too
      Set<? extends GraphSelection> selectionSet = m_graph.getSelection();
      if(!selectionSet.isEmpty()) {
        // Write out the selection state
        s.append("The current graph selection is: <br>");
        s.append(SelectionSample.convertSelectionStateToString(selectionSet));
      }
      m_status = s.toString();
      RequestContext.getCurrentInstance().addPartialTarget(m_outputFormatted);
    }
  }
}
```

The managed bean in the preceding example calls the `SelectionSample` class which is displayed in Example 23–29.

Example 23–29 Managed Bean for Custom Context Menu SelectionSample Class

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import javax.faces.model.SelectItem;
import oracle.adf.view.faces.bi.component.graph.DataSelection;
import oracle.adf.view.faces.bi.component.graph.GraphSelection;
import oracle.adf.view.faces.bi.event.graph.SelectionEvent;
import oracle.adf.view.faces.bi.model.KeyMap;
public class SelectionSample {
  public void selectionListener(SelectionEvent selectionEvent) {
    StringBuilder eventInfo = new StringBuilder();
    Set<? extends GraphSelection> selectionSet =
                          selectionEvent.getGraphSelection();
    eventInfo.append(convertSelectionStateToString(selectionSet));
    // Store on the selection string
    m_selectionInfo = eventInfo.toString();
  }
  /**
   * Returns selection state formatted with one selected data point per line.
   * @param selectionSet
   * @return
   */
  public static String convertSelectionStateToString
                    (Set<? extends GraphSelection> selectionSet) {
    StringBuilder selectionState = new StringBuilder();
    for(GraphSelection selection: selectionSet) {
      if(selection instanceof DataSelection) {
        DataSelection ds = (DataSelection) selection;
        Set seriesKeySet = ds.getSeriesKey().keySet();
        for(Object key : seriesKeySet) {
          selectionState.append(key).append(": ").
```

```
                                  append(ds.getSeriesKey().get((String)key));
            }
            List<KeyMap> groupKeys = ds.getGroupKeys();
            for(KeyMap groupKey : groupKeys) {
              Set groupKeySet = groupKey.keySet();
              for(Object key : groupKeySet) {
                selectionState.append("; ").append(key).append(": ").
                                        append(groupKey.get((String)key));
              }
            }
            selectionState.append("<br>");
          }
        }
        return selectionState.toString();
      }
      private String m_selectionInfo = "Select a marker to see information here.";
      public String getSelectionInfo() {
        return m_selectionInfo;
      }
      private String graphType = "bubbleGraph";
      public String getGraphType() {
          return graphType;
      }
      public void setGraphType(String type) {
          graphType = type;
      }
      private List graphList;
      public List getGraphList() {
          graphList = new ArrayList();
          SelectItem graph = new SelectItem("bubbleGraph", "Bubble Graph");
          graphList.add(graph);
          graph = new SelectItem("scatterGraph", "Scatter Graph");
          graphList.add(graph);
          return graphList;
      }
    }
```

24

Using Gauge Components

This chapter describes how to use the ADF Data Visualization `gauge` component to display data in gauges 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 use data controls to create gauges. For more information, see the "Creating Databound Graph and Gauge Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

This chapter includes the following sections:

24.1 About the Gauge Component

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 a graph. 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, or in a vertical, horizontal, or grid layout as a *gauge set*.

> **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, status meter, vertical status meter, and LED. All gauge types can display a title, bottom label, data label, and legend.

24.1.1 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: Indicates its metric along a configurable arc value axis. This is the default gauge type. Dial gauges can display as a simple gauge, a gauge with thresholds, or as a set of dial gauges.

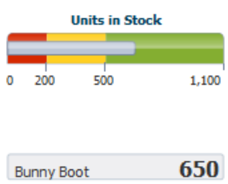
  Figure 24–1 shows a dial gauge with thresholds indicating a beginner's ski boot stock level within an acceptable range.

Figure 24–1 Dial Gauge with Thresholds

- Status Meter: Indicates the progress of a task or the level of some measurement along a horizontal rectangular bar. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Status meter gauges can display as a simple gauge, a gauge with thresholds, or as a set of status meter gauges.

  Figure 24–2 shows the Bunny Boot stock level using a status meter gauge.

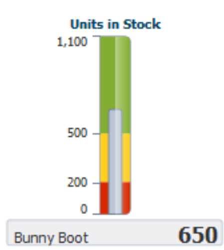
Figure 24–2 Status Meter Gauge with Thresholds



- Status Meter (vertical): Indicates the progress of a task or the level of some measurement along a vertical rectangular bar. Vertical status meter gauges can display as a simple gauge, a gauge with thresholds, or as a set of vertical status meter gauges.

  Figure 24–3 shows the Bunny Boot stock level using a vertical status meter gauge.

Figure 24–3 Vertical Status Meter Gauge with Thresholds



- LED (light-emitting diode): Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of graphics are available for LED gauges, such as round or rectangular shapes that use color to indicate status, and triangles or arrows that indicate good (up), fair (left- or right-pointing), or poor (down) states in addition to a color indicator. LED gauges can also display as a gauge set.

  Figure 24–4 shows the Bunny Boot stock level using a LED bulb indicator using color to indicate status.

Figure 24–4 LED Bulb Gauge

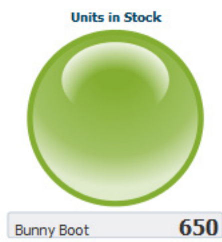
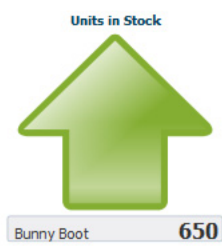


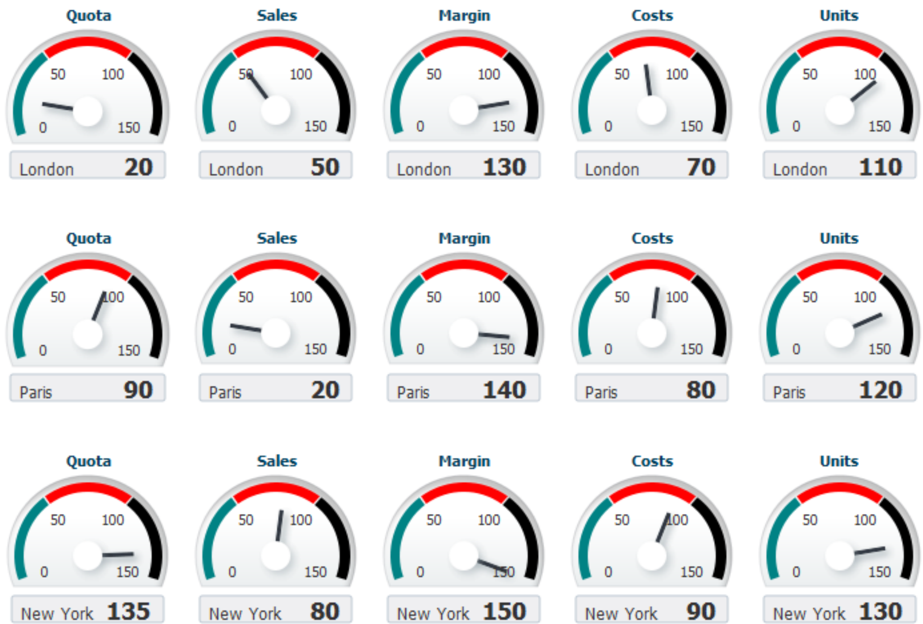
Figure 24–5 shows the same stock level using a LED arrow.

Figure 24–5 LED Arrow Gauge



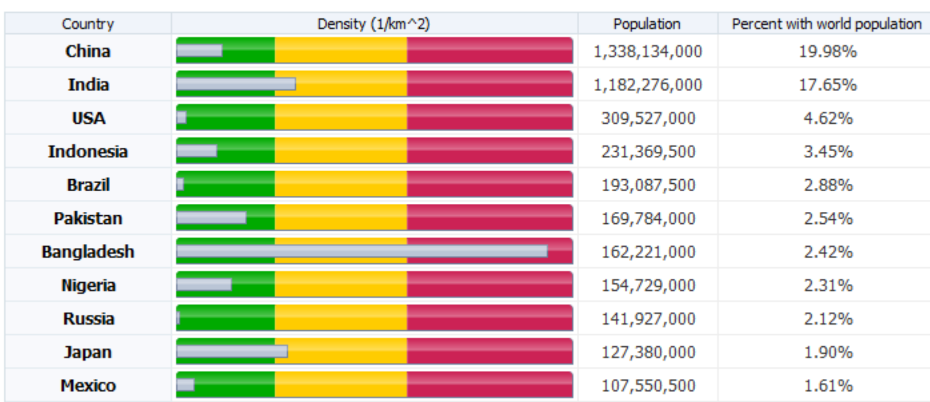
All gauge types can be displayed as a set of gauges in a built-in grid layout. Gauge sets are useful when displaying individual values for a group of related items. Figure 24–6 shows a gauge set comparing performance measures across three cities.

Figure 24–6 Gauge Set Comparing Performance Across Cities



Horizontal status meter and LED gauges are well-suited for display in table cells where users can see and compare them alongside related information such as labels, links, and icons. Figure 24–7 shows a table comparing the population density in countries with the highest population in 2010.

Figure 24–7 Horizontal Status Meter Gauges Displayed in Table



Country	Density (1/km^2)	Population	Percent with world population
China		1,338,134,000	19.98%
India		1,182,276,000	17.65%
USA		309,527,000	4.62%
Indonesia		231,369,500	3.45%
Brazil		193,087,500	2.88%
Pakistan		169,784,000	2.54%
Bangladesh		162,221,000	2.42%
Nigeria		154,729,000	2.31%
Russia		141,927,000	2.12%
Japan		127,380,000	1.90%
Mexico		107,550,500	1.61%

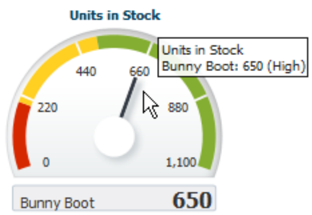
24.1.2 End User and Presentation Features of Gauge Components

To understand how gauges are used and can be customized, it may be helpful to review these elements and features:

- Display elements including:
  - Gauge and gauge set backgrounds
  - Gauge frames
  - Dial gauge plot area
  - Indicators and indicator bars

  - Gauge top, bottom, and metric labels
  - Thresholds and labels
  - Legends
  - Tick marks and labels
- Tooltips: A tooltip of contextual information automatically displays when a user moves a cursor over the plot area, indicator, or threshold region. Figure 24–8 shows the indicator tooltip for a dial gauge.

Figure 24–8 Indicator Tooltip for Dial Gauge



24.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: You can make your gauge components accessible. For more information, see Chapter 33, "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 Chapter 31, "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 Appendix D, "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 the "Creating Databound Graph and Gauge 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 Section 22.2, "Common Functionality in Data Visualization Components."

24.2 Using the Gauge Component

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 **Gauge Data** attributes category in the Properties window. It can also be specified through data controls or through the `tabularData` attribute of the `gauge` tag. This is the only required data for a gauge. The number of metric values supplied affects whether a single gauge is displayed or a series of gauges are displayed in a gauge set.
- 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 **Gauge Data** attributes category in the Properties window for the `gauge` tag.
- Threshold: 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 Section 24.3.1, "How to Configure Gauge Thresholds."

24.2.1 Configuring Gauges

The properties for the `gauge` component are sufficient to produce a gauge, but you can also add and configure child components or supported facets to customize the display and behavior of the gauge or gauge set. The prefix `dvt:` occurs at the beginning of each gauge component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library. You can configure gauge child components and supported facets in the following areas:

- Gauge display elements:
  - Gauge background (`gaugeBackground`) and gauge set background (`gaugeSetBackground`): Bounded area behind the gauge or gauge set.
  - Gauge frame (`gaugeFrame`): Refers to the decorative frame that encloses the plot area on dial gauges.
  - Plot area (`gaugePlotArea`): Indicates the graphical representation of the metric value of the gauge.

- Indicator (`indicator`): Points to the value that is plotted in a dial gauge, typically in the form of a line or an arrow.
- Indicator bar (`indicatorBar`): The inner rectangle in a status meter gauge.
- Indicator base (`indicatorBase`): The circular base of a line or needle style indicator in a dial gauge.
- Gauge labels:
  - Top label (`topLabel`): Shows the gauge title appearing at the top or inside of a gauge. You can configure an upper label frame (`upperLabelFrame`) for this label to specify border color and fill color. Turn off the default title separator when using this frame.
  - Bottom label (`bottomLabel`): Refers to an optional label that appears below or inside the gauge. By default, displays the label for the data row. You can configure a lower label frame (`lowerLabelFrame`) for this label to specify border color and fill color.
  - Metric label (`metricLabel`): Shows the value of the metric that the gauge is plotting in text.
- Thresholds and legend: Use a threshold set (`thresholdSet`) to specify the threshold sections (threshold) for the metrics of a gauge. You can create an unlimited number of thresholds for a gauge.

  A legend displays a description of the threshold set with the color and the name or range of each threshold. Legend elements include legend area (`gaugeLegendArea`), text (`gaugeLegendText`), and title (`gaugeLegendTitle`).
- Tick marks (`tickMark`): Refers to the markings along the value axis of the gauge. These can identify regular intervals, from minimum value to maximum value, and can also indicate threshold values. Tick marks can specify major increments that may include tick mark labels (`tickLabel`) or minor increments.
- Context menus (`bodyContextMenu` facet): Use this facet to support a single `af:popup` component containing a context menu that will be shown on the JSP or JSPX page when you right-click on any non selectable object within the component. The `af:popup` must contain an `af:menu` to display the context menu. Facets on facelet pages support multiple `af:popup` components.

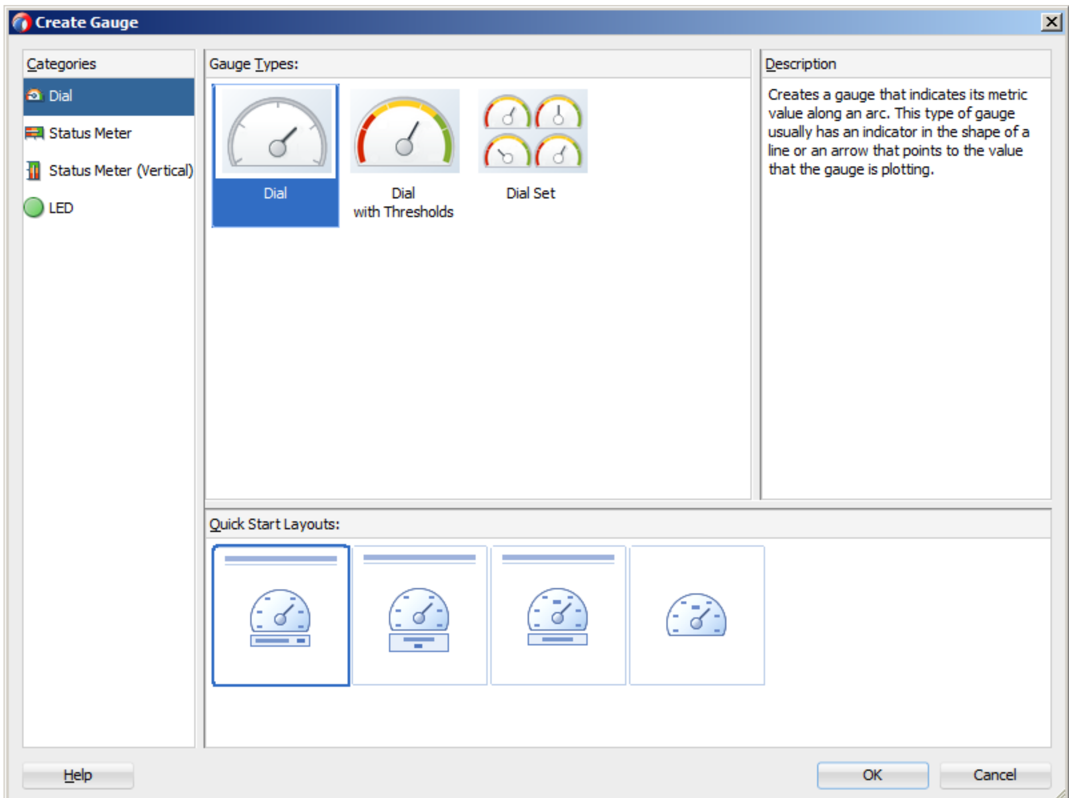
- Data values: Format categorical and numeric data values with standard ADF converters. For more information, see Section 24.5, "Formatting Numeric Data Values in Gauges."
- Interactivity: Use a shape attributes set (`shapeAttributesSet`) to configure behavior properties for gauge child elements. For example, the `alt` text of a gauge plot area can be displayed as a tooltip when the user moves the mouse over that area at runtime. For more information, see Section 24.6.3, "How to Add Interactivity to Gauges."
- Custom shapes: You can use a set of prebuilt custom styles for gauges, or specify a vector graphics file that is used for output by setting the `customShapesPath` attribute. For more information, see Section 24.7, "Using Custom Shapes for Gauges."
- Image formats: Gauges support the following image formats: HTML5, Flash, and PNG. By default, gauges in new applications display in HTML5, but you can change the default image format. You can also disable Flash across your application or customize its behavior on client platforms. For more information,

see Section 24.2.5, "What You May Need to Know About Gauge Image Formats."

24.2.2 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. You can also specify a quick start layout of the gauge's title and legend. Figure 24–9 shows the Create Gauge dialog with the dial gauge type selected.

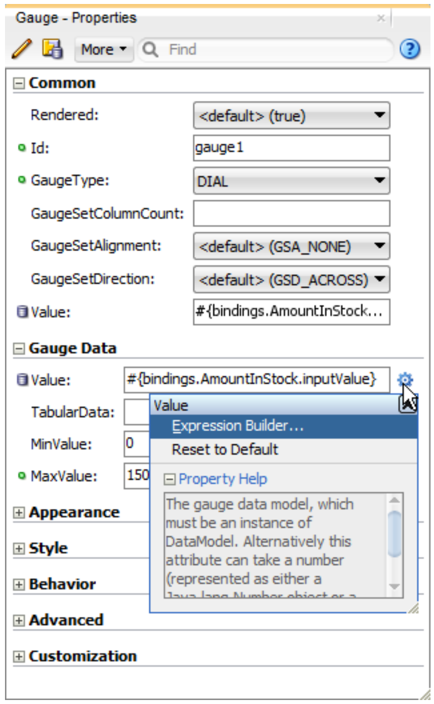
Figure 24–9 Create Gauge Dialog for Gauges



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 24–10 shows the Property menu for a gauge component `value` attribute.

Figure 24–10 Gauge Component Value Attribute Property Menu



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 the "Creating Databound Graph and Gauge 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 gauge attributes and gauge child components can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

To add a gauge to a page:

1. In the ADF Data Visualizations page of the Components window, from the Graph and Gauge panel, drag and drop a **Gauge** onto the page to open the Create Gauge dialog in the Components window.
2. Use the dialog to select the gauge category and type, and the quick start layout for display of gauge title, legend, and labels. For help with the dialog, press F1 or click **Help**.
3. In the Create Gauge dialog, click **OK** to add the gauge to the page.
4. In the Properties window, view the attributes for the gauge or gauge set. Use the **Component Help** button to display the complete tag documentation for the `gauge` component.

5. Expand the **Gauge Data** section. Specify data values for the gauge by setting the value in these fields:

   - **Value**: For a single gauge, specify the data model, which can be an attribute from a data control or an instance of `DataModel`, using an EL Expression. Alternatively, set a metric numeric value as either a `Java.lang.Number` object or a `String`.
   - **TabularData**: For a gauge set, specify a tabular data set as a `Java.util.List` object. For more information, see Section 24.2.4, "How to Create a Gauge Using Tabular Data."
   - **MinValue** and **MaxValue**: Optionally, set the lowest and greatest values on the gauge axis. These values are set automatically if not specified.

6. Expand the **Appearance** section. Specify display attributes by setting the value in these fields:

   - **LedStyle**: If you wish to change the shape of the LED gauge, use the dropdown list to select any of the following valid values: `LS_DOT`, `LS_ARROW`, `LS_RECTANGLE`, or `LS_TRIANGLE`. You can also use the `LS_CUSTOM` value if you wish to specify a custom image.
   - **ThresholdDialStyle**: If you wish to change the default style (`TDS_SEGMENTS`) of thresholds in dial gauges, use the dropdown list to either of the following valid values: `TDS_PIE_FILL`, `TDS_RING_FILL`.
   - **AngleExtent**: Use to specify the range of degrees that sweeps through angles other than the standard 220-degree arc in a dial gauge.
   - **CustomShapesPath**: Use to specify the path to the custom shape definition file. For more information, see Section 24.7, "Using Custom Shapes for Gauges."
   - **ShortDesc**: Enter a description of the gauge. This description is accessed by screen reader users.
   - **AnimationOnDisplay**, **AnimationOnDataChange**, **AnimationDuration** (Animation sub-section): Use one or more of these attributes to set animation effects for the gauge. For more information, see Section 24.6.4, "Animating Gauges."

The gauge will display on the client in the HTML5 image format if the client supports it. For more information about gauge image formats, see Section 24.2.5, "What You May Need to Know About Gauge Image Formats."

24.2.3 What Happens When You Add a Gauge to a Page

When a gauge component is inserted into a JSF page using the Components window, a set of child tags that support customization of the gauge is automatically inserted. Example 24–1 shows the code inserted in the JSF page for a dial gauge with the quick-start layout selected in the Create Gauge dialog in Figure 24–9.

Example 24–1 Gauge Sample Code

```
<dvt:gauge id="gauge1" gaugeType="DIAL">
  <dvt:gaugeBackground>
    <dvt:specialEffects/>
  </dvt:gaugeBackground>
  <dvt:gaugeFrame/>
  <dvt:indicator/>
  <dvt:indicatorBase/>
```

```
  <dvt:gaugePlotArea/>
  <dvt:tickLabel/>
  <dvt:tickMark/>
  <dvt:topLabel/>
  <dvt:bottomLabel/>
  <dvt:metricLabel position="LP_WITH_BOTTOM_LABEL"/>
</dvt:gauge>
```

24.2.4 How to Create a Gauge Using Tabular Data

A gauge set is created when a grid of data is used for the gauge component. The `tabularData` attribute of a gauge component lets you specify a list of values that the gauge uses to create a grid and to populate itself. To create a gauge using tabular data you must store the data in a method in the gauge's managed bean and then use the gauge component's `tabularData` attribute to reference the data. For more information about creating and using managed beans, see Section 3.6, "Creating and Using Managed Beans."

When you provide only the metric value through the `tabularData` attribute, each value in the grid is represented by a separate gauge. In this case you must specify any desired thresholds and minimum or maximum values through the Properties window.

For example, the table in Figure 24–11 has five columns: Quota, Sales, Margin, Costs, and Units, and three rows: London, Paris, and New York. This data produces a gauge set with five gauges in each row similar to the gauge set in Figure 24–6 and lets you compare values such as sales across the three cities.

Figure 24–11 Comparison of Annual Results

Desired Specs	Quota	Sales	Margin	Costs	Unit
London	20	50	130	70	110
Paris	90	20	140	80	120
New York	135	80	150	90	130

In a managed bean, the structure of the list of tabular data consists of a three-member `Object` array for each data value to be passed to the gauge. The members of each array must be organized as follows:

- The first member (index 0) is the column label of the data value in the grid. This is generally a `String`.
- The second member (index 1) is the row label of the data value in the grid. This is generally a `String`.
- The third member (index 2) is the data value, which is usually `Double`.

Example 24–2 shows code in a managed bean method that creates the list of tabular data required for the gauge that compares annual results for three cities displayed in Figure 24–11.

Example 24–2 Managed Bean Method to Create a List of Tabular Data for Annual Results

```
public List getGaugeData()
{
    ArrayList list = new ArrayList();
    String[] rowLabels  = new String[] {"London", "Paris", "New York"};
    String[] colLabels  = new String[] {"Quota", "Sales", "Margin", "Costs",
"Units"};
    double [] [] values = new double[][]{
        {20, 90, 135},
        {50, 20, 80},
```

```
        {130, 140, 150},
        {70, 80, 90},
        {110, 120, 130}
        };
    for (int c = 0; c < colLabels.length; c++)
    {
     for (int r = 0; r < rowLabels.length; r++)
       {
        list.add (new Object [] {colLabels[c], rowLabels[r],
            new Double (values [c][r])});
       }
    }
    return list;
}
```

To provide the metric value and threshold, minimum, and maximum values, use data specification to set the columns or rows of data through the gauge's `value` attribute.

For example, the data in Figure 24–12 provides the metric values and minimum, maximum, and threshold values for two cities. The data produces a gauge set of two gauges comparing sales results by desired specifications.

Figure 24–12 Comparison of Sales Results by Specification

Desired Specs	**Metric**	**Minimum**	**Maximum**	**Threshold1**	**Threshold2**
Name	Sales	Min	Max	Quota	Target
Boston	40	0	100	30	50
Chicago	60	0	80	35	70

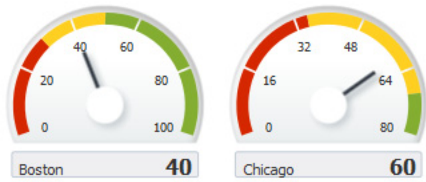
Example 24–3 shows code in a managed bean method that creates the list of tabular data required for the gauge that compares sales results by specification for two cities displayed in Figure 24–12.

Example 24–3 Managed Bean Method to Create a List of Tabular Data for Sales Results

```
public GaugeDataModel getGaugeData()
{
// Set up values for data source
    Object[] colLabels = { "Sales", "Min", "Max", "Quota", "Target" };
    Object[] rowLabels = { "Boston", "Chicago" };
    Double[][] values = { {40.0, 60.0}, {0.0,0.0}, {100.0,80.0}, {30.0,35.0},
{50.0,70.0} };
    LocalXMLDataSource dataSource =
                       new LocalXMLDataSource(colLabels, rowLabels, values);
// Set up values for data specification
    DataSpecification dataSpec = new DataSpecification();
    dataSpec.setMetric("Sales");
    dataSpec.setMinimum ("Min");
    dataSpec.setMaximum ("Max");
    ArrayList threshdata = new ArrayList ();
    threshdata.add ("Quota");
    threshdata.add ("Target");
    dataSpec.setThresholds(threshdata);
    return new GaugeDataModel(dataSource, dataSpec);
}
```

Figure 24–13 shows the gauge set that is rendered on the page if you add the method in Figure 24–3 to a dial gauge set.

Figure 24–13 Gauge Set Using Data Specification to Compare Sales Results



Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge set on your page. If you do not, follow the instructions in this chapter to create a gauge set. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To create a gauge set using tabular data from a managed bean:

1. Add the method to create tabular data to the gauge's managed bean. For additional help, see Section 3.6, "Creating and Using Managed Beans."
2. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.
3. In the Properties window, expand the **Gauge Data** section.
4. From the **TabularData** or **Value** attribute dropdown menu, choose **Expression Builder**.
5. Locate the managed bean node and expand it.
6. Select the method that creates the list of tabular data and click **OK**.

   The Expression is created.

   For example, if the name of the managed bean is `sampleGauge` and the name of the method that creates the list of tabular data is `getGaugeData`, the Expression Builder generates the code `#{sampleGauge.gaugeData}` as the value for the `tabularData` attribute of the `gauge` component.
7. Use the Properties window to configure any desired properties for the gauge.

   For example, if you used the sample method in Example 24–2 to create a gauge set displaying five gauges in each row, set the **GaugeSetColumnCount** in the **Common** section to 5.

24.2.5 What You May Need to Know About Gauge Image Formats

Gauges support the following image formats: HTML5, Flash, and PNG. The image format used depends upon the application's settings and the client's environment.

You can configure your application to use a specific image format by setting or changing the following parameters:

- `oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT`

This context initialization parameter is automatically added to `web.xml` for all new applications and defaults to HTML5. For more information, see Section A.2.3.28, "Graph and Gauge Image Format."

- Skin style

  Gauges will be displayed in the HTML5 image format when using the Skyros skin. New applications default to this skin. For more information about skinning and styles, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

- `flash-player-usage`

  You can disable 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 Section A.4.3, "Configuring Flash as Component Output Format."

If the specified image 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.
- Portable Network Graphics (PNG) output format. A PNG output format is used also when printing gauges. Although static rendering is fully supported when using a PNG output format, certain interactive features are not available including:
  - Animation
  - Context menus
  - Popup support
  - Interactivity

All image formats for gauges support bi-directional locales. Support includes text strings containing bi-directional characters, label positions, legend display, and gauge set display.

24.2.6 How to Add Gauges to Tables

You can display gauges in table cells where users can see and compare them alongside related information. The immediate children of an ADF `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.

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.

Example 24–4 shows sample code for displaying gauges in the Oracle ADF table component shown in Figure 24–7. In this example, the table's `var` property is `testvar`, and the table's value is stored in a managed bean named `gaugeData`. The gauge's metric is set in the **value** field to `#{testvar.density}`.

Example 24–4 Gauge Component Stamped in Table Column

```
<af:table summary="table" value="#{gaugeData.gaugeTableData}" var="testvar"
```

```
            rowBandingInterval="0" id="t1" columnStretching="last"
            inlineStyle="height:400px" styleClass="AFStretchWidth">
    <af:column rowHeader="true" sortable="false" headerText="Country"
               align="center" id="c1" width="120"
               inlineStyle="font-weight:bold; font-size: 12px;">
      <af:outputText value="#{testvar.name}" id="ot1"/>
    </af:column>
    <af:column sortable="true" headerText="Density (1/km^2)"
               align="center" id="c2" width="300"
               sortProperty="density">
      <dvt:gauge shortDesc="Gauge" id="gauge77" gaugeType="STATUSMETER"
                 dynamicResize="DYNAMIC_SIZE"
                 value="#{testvar.density}"
                 inlineStyle="height:22px" styleClass="AFStretchWidth"
                 minValue="0.0" maxValue="1200.0">
        <dvt:indicatorBar/>
        <dvt:thresholdSet>
          <dvt:threshold fillColor="#00aa00" thresholdMaxValue="300"/>
          <dvt:threshold fillColor="#ffcc00" thresholdMaxValue="700"/>
          <dvt:threshold fillColor="#cc2255"/>
        </dvt:thresholdSet>
        <dvt:topLabel position="LP_NONE"/>
        <dvt:bottomLabel position="LP_NONE"/>
        <dvt:metricLabel position="LP_NONE"/>
      </dvt:gauge>
    </af:column>
    <af:column sortable="false" width="100" inlineStyle="font-size: 12px;"
               headerText="Population" align="center" id="c3">
      <af:outputText value="#{testvar.population}" id="ot3">
        <af:convertNumber type="number" pattern="#,##0"/>
      </af:outputText>
    </af:column>
    <af:column sortable="false" width="100" inlineStyle="font-size: 12px;"
               headerText="Percent with world population" align="center" id="c4">
      <af:outputText value="#{testvar.percent}" id="ot4">
        <af:convertNumber type="percent" pattern="#.00%"/>
      </af:outputText>
    </af:column>
  </af:table>
```

Example 24–5 shows the sample code for the gaugeData managed bean. In this example, the class is named GaugeTableData.

Example 24–5 Managed Bean Code for Gauge in Table Example

```
public class GaugeTableData {
  private List<CountryData> _gaugeTableData;
  public GaugeTableData() {
    ArrayList<CountryData> list = new ArrayList<CountryData>(11);
    list.add(new CountryData("China", 1338134000,139.434));
    list.add(new CountryData("India", 1182276000, 359.654));
    list.add(new CountryData("USA", 309527000,32.145));
    list.add(new CountryData("Indonesia", 231369500,121.481));
    list.add(new CountryData("Brazil", 193087500, 22.676));
    list.add(new CountryData("Pakistan", 169784000, 211.19));
    list.add(new CountryData("Bangladesh", 162221000,1126.55));
    list.add(new CountryData("Nigeria", 154729000, 167.498));
    list.add(new CountryData("Russia", 141927000,8.301));
    list.add(new CountryData("Japan", 127380000,337.097));
    list.add(new CountryData("Mexico", 107550500,54.923));
    _gaugeTableData = list;
```

```
  }
  public List<CountryData> getGaugeTableData(){
      return _gaugeTableData;
  }
  public static class CountryData {
    private String _name;
    private long _population;
    private double _percent;
    private double _density;
    private static long worldPopulation = 6697254000L;
    public CountryData(String name, int population, double density) {
      _name = name;
      _population = population;
      _density = density;
      _percent = (double)population / (double)worldPopulation;
    }
    public String getName() {
      return _name;
    }
    public double getPercent() {
      return _percent;
    }
    public long getPopulation() {
      return _population;
    }
    public double getDensity() {
      return _density;
    }
  }
}
```

When configuring gauges in table cells, use these guidelines to improve usability:

- Use horizontal status meters, dial or LED gauges. Avoid the use of vertical status meters because the table rows would have to be very tall to accommodate them.
- Make gauges as small as possible while maintaining legibility.
- To maximize use of space, use table column and row headers to describe gauges, rather than using gauge titles or bottom labels.
- Use only one type of gauge per column or row, and use the same axis values and thresholds.
- Display horizontal status meters in columns, and dial and LED gauges in rows or columns.
- Avoid displaying so many gauges that users must scroll to see them all.

If your application uses the Fusion technology stack, then you can use data controls to create a databound ADF table and add the gauge during table creation. For more information, see the "Creating Databound Graph and Gauge 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 gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You must complete the following tasks:

1. Create the managed bean that contains the data for the table and gauge.

   To duplicate the table displayed in Figure 24–7, create a managed bean named `gaugeData` and add the code in Example 24–5. For help with managed beans, see Section 3.6, "Creating and Using Managed Beans."

2. Create the ADF table.

   Example 24–4 shows the code used to display the table in Figure 24–7. By default, the gauge component is not listed as an option during table creation. Configure the column that will contain the gauge as any valid component. You will delete the component when you add the gauge to the table.

   If you need help creating the table, see Section 12.3.4, "How to Display a Table on a Page."

To add gauges to a table:

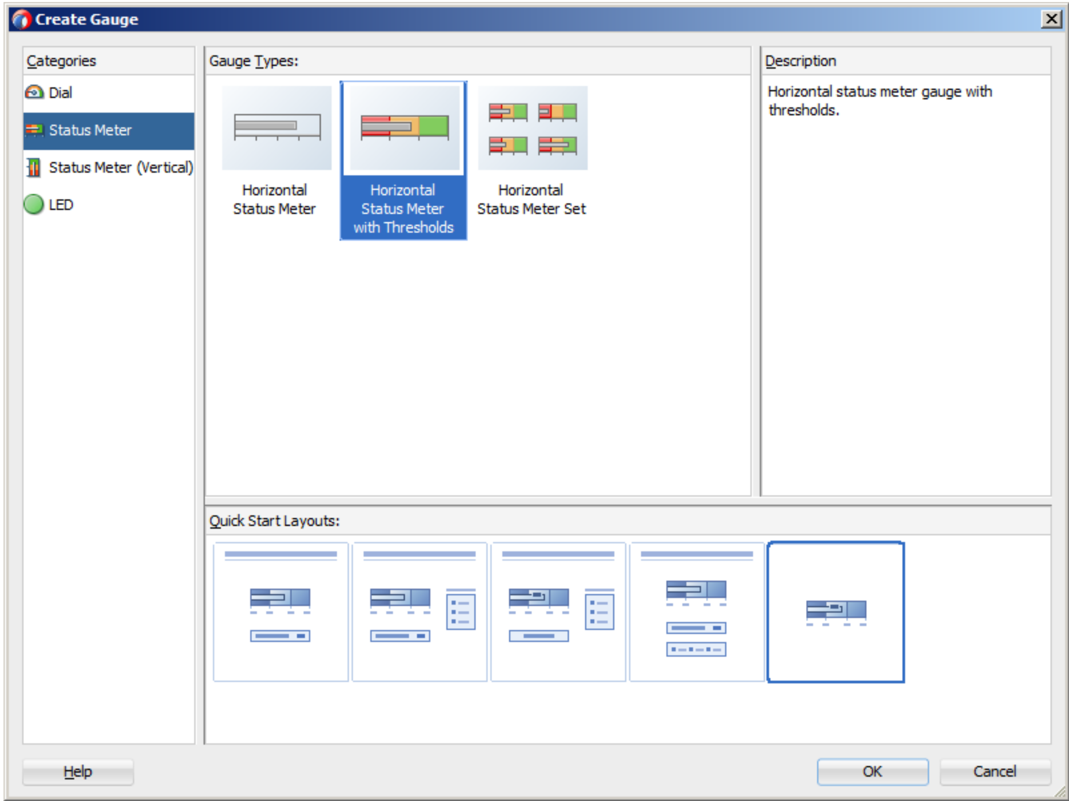
1. In the Structure window, expand **af:table**.
2. Right-click the component that the gauge will replace and choose **Delete**.

   For example, if you configured the table column to use `af:outputText`, right-click it and choose **Delete**.

3. Right-click the column that will contain the gauge and choose **Insert Inside Column > ADF Data Visualizations > Gauge**.
4. In the Create Gauge dialog, choose the gauge type and a quick start layout that does not include labels.

   Figure 24–14 shows the Create Gauge dialog for the gauge displayed in Figure 24–7.

Figure 24–14 Create Gauge Dialog for Horizontal Status Meter in Table



5. Click **OK** to add the gauge to the table.
6. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
7. In the Properties window, in the **Value** field, enter the metric value that the gauge is to display.

   For example, to set the gauge to the value used by the gauge in Figure 24–7, enter: `#{testvar.density}`.
8. If you selected a quick start layout that includes thresholds, in the Properties window, enter values for each of the thresholds.

   For help with configuring thresholds, see Section 24.3.1, "How to Configure Gauge Thresholds."
9. Format the gauge style elements as needed.

   For example, the code example in Example 24–4 sets values for height and size to the following:

   ```
   inlineStyle="height:22px" styleClass="AFStretchWidth"
   ```
   For help with configuring gauge style elements, see Section 24.4, "Formatting Gauge Style Elements."

24.3 Configuring Gauge Display Elements

You can customize gauge display elements including thresholds, labels, indicators, and tick marks.

24.3.1 How to Configure Gauge Thresholds

Thresholds are numerical data values in a gauge that highlight a particular range of values. Thresholds must be values between the minimum and the maximum value for a gauge. The range identified by a threshold is filled with a color that is different from the color of other ranges.

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 the "Creating Databound Graph and Gauge Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

After the gauge is created, you can also specify static threshold values by configuring a `thresholdSet` child component that wraps an unlimited number of `threshold` child components in a gauge. If threshold values are supplied in both the data collection and in `threshold` components, the gauge honors the values in the `threshold` components. You can also add a legend to the threshold set to provide a visual definition of the threshold values for the user.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

Note: If you create a gauge using a gauge type with thresholds in the Create Gauge dialog, a **dvt:thresholdSet** component and **dvt:threshold** component children are automatically added to the **dvt:gauge** component in the Structure window.

If you choose a quick start layout that includes a legend, a **dvt:gaugeLegendTitle** and **dvt:gaugeLegendArea** are also automatically added to the **dvt:gauge** component in the Structure window.

To add static thresholds to a gauge:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **ADF Data Visualizations** > **Threshold Set**.
2. Right-click the **dvt:thresholdSet** component and choose **Insert Inside Threshold Set** > **Threshold**.
3. In the Properties window, set the following attributes:
   - **ThresholdMaxValue**: Specify the maximum value for the threshold section you are configuring. Values can be an integer or an EL expression.

Note: For the final threshold, the maximum value of the gauge is used as the threshold maximum value regardless of any entry you make in the **ThresholdMaxValue** attribute for the final threshold.

- **FillColor** and **BorderColor**: Optionally, specify a RGB value for the fill color and border color respectively for the threshold section you are configuring. You can also change the color from opaque to transparent. For more information, see Section 24.4.2, "Specifying Transparency for Gauge Elements."
- **Text**: Optionally, specify the text to be displayed in the legend to identify this threshold. You can also bind the text to a text resource. For more information about text formatting, see Section 24.4.3, "How to Format Gauge Text." For more information about text resources, see Section 24.4.4, "How to Specify a Gauge Text Resource."

4. Repeat Step 2 and Step 3 to create each threshold in the gauge from the lowest minimum value to the highest maximum value.

Note: You have the option of adding any number of thresholds to gauges. However, arrow and triangle LED gauges support thresholds only for the three directions to which they point.

To add an optional legend to the gauge's threshold set:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **ADF Data Visualizations** > **Legend Area**.
2. In the Properties window, set the following attributes:
   - **Position**: Specify the legend's position in relation to the gauge.

     By default, the legend's position is set to `LAP_BOTTOM`, and the legend displays below the gauge. Use `LAP_LEFT`, `LAP_RIGHT`, or `LAP_TOP` to display the legend on the left, right, or top of the gauge, respectively.
   - **BorderColor** and **FillColor**: Optionally, set a border and fill color for the legend.

     Specify the RGB value in hexadecimal notation (`#000000`, for example), or choose **Edit** from the attribute's dropdown menu to set the color with the Edit Border Property dialog.
3. To add a title to the legend:
   1. In the Structure window, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **ADF Data Visualizations** > **Legend Title**.
   2. Right-click the **dvt:gaugeLegendTitle** component and choose **Go to Properties**.
   3. In the **Text** field, enter the title for the legend.
4. To configure the legend text:
   1. In the Structure window, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **ADF Data Visualizations** > **Legend Text**.
   2. Right-click the **dvt:gaugeLegendText** component and choose **Go to Properties**.
   3. In the **MinLength** field, enter the minimum length of text to display when truncation is needed to fit text into the available space.
   4. In the **NumberType** field, use the attribute's dropdown menu to set the legend's number type to **NT_PERCENT** (default) or **NT_NUMBER**.

Note: The **gaugeLegendText** component is only required to define a `minLength` or to change the default `numberType` to NT_NUMBER.

24.3.2 How to Customize Gauge Labels

By default gauges display a metric label and optional top and bottom labels using the child components `metricLabel`, `topLabel`, and `bottomLabel`. You can customize the display and positioning of each label, as well as control the fill and border colors of the optional top and bottom gauge label frames.

The categorical data value represented by the top or bottom label can also be customized using an `attributeFormat` tag and ADF Faces converter tags to format percents, scale numbers, control the number of decimal places, placement of signs, and so on. For more information, see Section 24.5.1, "How to Format Numeric Data Values in Gauges."

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

Note: When you add a gauge to your page, gauge child label components are automatically added to the `gauge` component. The default location, if any, of each gauge label is specified in the `position` attribute of the component, based on the choice you make for a quick start layout option in the Create Gauge dialog.

To customize a gauge label:

1. In the Structure window, select the gauge child label component (**dvt:metricLabel**, **dvt:topLabel**, or **dvt:bottomLabel**) you wish to configure.
2. In the Properties window, set the following attributes:
   - **Position**: Use to specify the location, if any, of the gauge label. Valid values include:
     - `LP_NONE`: No label is displayed.
     - `LP_INSIDE_GAUGE`: Label is displayed inside the plot area of the gauge. Labels are horizontally centered across a gauge. When set to this value, top and bottom labels are also vertically centered inside the plot area. This is a good choice for the LED gauge.
     - `LP_INSIDE_GAUGE_RIGHT` and `LP_INSIDE_GAUGE_LEFT`: Metric label is displayed either to the right or left of the plot area.
     - `LP_ABOVE_GAUGE`: The default value for the top label. Displays the label above the gauge.

- LP_BELOW_GAUGE: The default value for the bottom label. Displays the label below the gauge. If the position of both the bottom and metric labels are set to this value, then both labels are displayed below the gauge. However, the bottom label is displayed above the metric label.
- LP_WITH_BOTTOM_LABEL: The default value for the metric label. Displays the label beside the bottom label.

- **Text**: The text displayed in the top or bottom label. In the attribute menu, enter the text to display or choose Select Text Resources for a dialog to associate the text with application text resources. For more information, see Section 24.4.4, "How to Specify a Gauge Text Resource."
- **NumberType**, **Scaling**, and **AutoPrecision**: Available only for metric labels. Use these attributes to configure the display of numeric data values in the gauge. For more information, see Section 24.5.2, "What You May Need to Know About Automatic Scaling and Precision."

3. If you wish to configure the text font used in the gauge label, do the following:

   a. In the Structure window, right-click the gauge child label component (**dvt:metricLabel**, **dvt:topLabel**, or **dvt:bottomLabel**) you wish to configure and choose **Insert Inside *Label* > ADF Data Visualizations > Font**.

   b. Right-click the **dvt:gaugeFont** node and choose **Go to Properties**.

   c. In the Properties window, set the attributes for the font. For more information, see Section 24.4.3, "How to Format Gauge Text."

4. If you wish to configure a frame around the top or bottom gauge label, do the following:

   a. In the Structure window, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **Upper Label Frame** or **Lower Label Frame**.

   b. Right-click the **dvt:upperLabelFrame** or **dvt:lowerLabelFrame** and choose **Go to Properties**.

   c. In the Properties window, specify a RGB value for the **FillColor** and **BorderColor** attributes for the label frame you are configuring. You can also change the color from opaque to transparent. For more information, see Section 24.4.2, "Specifying Transparency for Gauge Elements."

24.3.3 How to Customize Gauge Indicators and Tick Marks

Gauges use a graphic to indicate the precise gauge value. By default gauges display a line for dial gauges using the child component `indicator`, and a bar inside status meter or vertical status meter gauges using the child component `indicatorBar`. The child component `indicatorBase` is used to set the fill properties of the circular base of all indicators of a dial gauge. You can customize the appearance of gauge indicators.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

Note: When you add a gauge to your page, gauge child indicator components are automatically added to the `gauge` component based on the gauge type you chose in the Create Gauge dialog. LED gauges do not have indicators.

To customize the appearance of gauge indicators:

- For dial gauge indicators, do the following:
  1. In the Structure window, right-click the **dvt:indicator** node and choose **Go to Properties**.
  2. In the Properties window, set the following attributes:
     – **Type**: Identifies the kind of indicator: a line indicator (default), a fill indicator, or a needle indicator.
     – **BorderColor**: Specifies the color of the border of the indicator.
     – **FillColor**: Specifies the color of the fill for the indicator.
     – **UseThresholdFillColor**: Determines whether the color of the threshold area in which the indicator falls should override the specified color of the indicator.
  3. In the Structure window, right-click the **dvt:indicatorBase** node and choose **Go to Properties**.
  4. In the Properties window, set the following attributes:
     – **Rendered**: Indicates whether or not the indicator base is displayed. The default value is `true`.
     – **BorderColor**: Specifies the color of the border of the indicator.
     – **FillColor**: Specifies the color of the fill for the indicator.
- For status meter and vertical status meter gauges, do the following:
  1. In the Structure window, select the **dvt:indicatorBar** component.
  2. In the Properties window, set the following attributes:
     – **BorderColor**: Specifies the color of the border of the indicator.
     – **FillColor**: Specifies the color of the fill for the indicator.

Note: If you want to specify that the color of the threshold area in which the indicator bar falls should override the specified color of the indicator, add an `indicator` component to the gauge, and set its `UseThresholdFillColor` attribute to `true`.

Tick marks are incremental marks along the gauge value axis for dial, status meter, and vertical status meter gauges. LED gauges do not have tick marks. By default, gauges display tick marks using the gauge child `tickMark` component to specify the display, spacing, and color of major and minor tick marks.

The gauge child `tickLabel` component identifies major tick labels to specify the location of the labels (interior or exterior of the gauge), and the format for numbers displayed in the tick labels. Minor tick marks do not support labels.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

> **Note:** When you add a gauge to your page, gauge child tick mark and tick mark label components are automatically added to the `gauge` component based on the choices you make in the Create Gauge dialog. LED gauges do not have tick marks.

To customize the tick marks and tick labels for a gauge:

1. To customize the gauge's tick marks, in the Structure window, right-click the **dvt:tickMark** node and choose **Go to Properties**.
2. In the Properties window, set values for the following attributes:
   - **MajorIncrement** and **MinorIncrement**: Sets the distance between two major tick marks and two minor tick marks, respectively. If the value is less than zero for either attribute, the tick marks are not displayed.
   - **MajorTickColor** and **MinorTickColor**: Sets the hexidecimal color of major tick marks and minor tick marks, respectively.
   - **Content**: Specifies where tick marks occur within a gauge set. Valid values are any combination separated by spaces or commas including:
     * `TC_INCREMENTS`: Display tick marks in increments.
     * `TC_MAJOR_TICK`: Display tick marks for minimum, maximum, and incremental values.
     * `TC_MIN_MAX`: Display tick marks for minimum and maximum values.
     * `TC_METRIC`: Display tick marks for actual metric values.
     * `TC_NONE`: Display no tick marks.
     * `TC_THRESHOLD`: Display tick marks for threshold values.

     If you do not specify a value for **Content**, the tick mark display defaults to `TC_MIN_MAX TC_INCREMENTS` which will display tick marks for minimum, incremental, and maximum values.
3. To customize the gauge's tick labels, in the Structure window, right-click the **dvt:tickLabel** node and choose **Go to Properties**.
4. In the Properties window, set values for the following attributes:

- **Position**: By default, the dial gauge displays interior tick labels to provide a cleaner look when the gauge is contained entirely within the gauge frame. Because the tick labels lie within the plot area, the length of the tick labels must be limited to fit in this space. You can customize your gauge to use exterior labels by setting the value for this attribute to `TLP_EXTERIOR` from the default `TLP_INTERIOR`.
- **Content**: Specifies which tick marks in a gauge have labels. Valid values are any combination separated by spaces or commas including:
  - `TC_INCREMENTS`: Display tick labels in increments.
  - `TC_MAJOR_TICK`: Display tick labels for minimum, maximum, and incremental values.
  - `TC_MIN_MAX`: Display tick labels for minimum and maximum values.
  - `TC_METRIC`: Display tick labels for actual metric values.
  - `TC_NONE`: Display no tick labels.
  - `TC_THRESHOLD`: Display tick labels for threshold values.

  If you do not specify a value for **Content**, the tick label display defaults to `TC_MAJOR_TICK` which will display tick labels for minimum, incremental, and maximum values.
- **NumberType**, **Scaling**, and **AutoPrecision**: Available only for metric and tick labels. Use these attributes to configure the display of numeric data values in the gauge. For more information, see Section 24.5.1, "How to Format Numeric Data Values in Gauges."

24.4 Formatting Gauge Style Elements

You can customize the styling of gauges to change the initial size or a gauge, specify dynamic resizing to fit the presentation area of a gauge, and apply style elements. You can also use text formatting and text resource, and transparency in gauges.

24.4.1 How to Change Gauge Size and Apply CSS Styles

Gauges are displayed in a default size of 200 X 200 pixels. You can customize the size of a gauge or specify dynamic resizing to fit an area across different browser window sizes. When gauges are displayed in a horizontally or vertically restricted area, for example in a web page sidebar, the gauge is displayed in a small image size. Although fully featured, the smaller image is a simplified display.

You can customize the width and height of a gauge, and you can allow for dynamic resizing of a gauge based on changes to the size of its container. These two aspects of a gauge are interrelated in that they share the use of the gauge `inlineStyle` attribute.

You can also apply CSS styles such as `active`, `focus`, `hover`, `link`, and `visited` to use for a gauge.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To specify the size of a gauge:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.
2. In Properties window, expand the **Style** section. Specify the initial size of the gauge in the **InlineStyle** attribute. If you do not also provide for dynamic resizing of the gauge, then the initial size becomes the only display size for the gauge. For example,

   ```
   width:200px;height:200px
   ```

   If you are specifying dynamic resizing for the gauge, you can enter a fixed number of pixels or 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. If you want to specify dynamic resizing for the gauge, expand the **Behavior** section. From the **DynamicResize** attribute dropdown list, select `DYNAMIC_SIZE`.

To apply CSS styles to a gauge:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.
2. In the Properties window, expand the **Style** section and enter the name of the style class in the **StyleClass** field.

   For information about applying CSS styles, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

24.4.2 Specifying Transparency for Gauge Elements

You can specify that various elements of a gauge display a transparent color instead of the default opaque color by setting the `borderColor` and `fillColor` attributes on the gauge child components related to those elements. These color properties accept a 6 or 8 RGB hexidecimal value. When an 8-digit value is used, the first two digits represent transparency. For example, you can set transparency by using a value of `00FFFFFF`.

Any gauge child component that supports `borderColor` or `fillColor` attributes can be set to transparency. The following are examples of gauge child components that support transparency:

- `gaugeBackground`
- `gaugeFrame`
- `gaugePlotArea`
- `gaugeLegendArea`

24.4.3 How to Format Gauge Text

You can format the text in gauges using a `gaugeFont` component as a child for any of the gauge child components that represent titles and labels in a gauge:

- `bottomLabel`
- `metricLabel`
- `gaugeLegendText`
- `gaugeLegendTitle`
- `tickLabel`
- `topLabel`

The attributes of the `gaugeFont` component allow you to specify these font attributes for the gauge child element:

- `name`: Specifies the name of the font, for example `San Serif`.
- `size`: Specifies the font size in pixels, for example `11`.
- `color`: Specifies the color of the font. This color property accepts a 6 or 8 RGB hexidecimal value. When an 8-digit value is used, the first two digits represent transparency. For example, you can set transparency by using a value of `00FFFFFF`.
- `bold`: Specifies whether or not the font is bold. The default value is `FALSE`.
- `italic`: Specifies whether or not the text is in italics. The default value is `FALSE`.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

> **Note:** When you add a gauge to your page, gauge child components for titles or labels are automatically added to the `gauge` component, based on the choices you make in the Create Gauge dialog.

To specify a text font for a gauge title or label component:

1. In the Structure window, right-click the gauge child component for a title or label and choose **Insert Inside *Title*** or ***Label* > Font**.

   For example, to specify a text font for the gauge's metric label, choose **Insert Inside Metric Label > Font**. If the component is not available, right-click the **dvt:gauge** component and choose **Insert Inside Gauge** > **ADF Data Visualization** > ***gauge child title or label component***.

2. Right-click the **dvt:gaugeFont** node and choose **Go to Properties**.

3. In the Properties window, set values for one or more of the **dvt:gaugeFont** component attributes. Click **Help** or press F1 to display the complete tag documentation for the `dvt:gaugeFont` component.

You can also set the font 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 fonts 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 Section 24.4.5, "How to Set Gauge Styles Globally Using a Skin."

24.4.4 How to Specify a Gauge Text Resource

JDeveloper supports easy localization of data visualization components using the abstract class `java.util.ResourceBundle` to provide locale-specific resources. For those gauge child components that represent titles and labels in a gauge, you can associate a text resource referenced in an application resource bundle. For more information, see Chapter 32, "Internationalizing and Localizing Pages."

Before You Begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To specify a text resource for a gauge title or label component:

1. In the Structure window, select a gauge child label or title component and choose **Go to Properties**.

   For example, to access the properties for the gauge's top label, right-click **dvt:topLabel** and choose **Go to Properties**.

2. In the Properties window, in the **text** attribute menu, choose **Select Text Resources**.

3. In the Select Text Resources dialog, associate the component text with a text resource.

   For help with the dialog, click **Help** or press F1.

24.4.5 How to Set Gauge Styles Globally Using a Skin

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-gauge
- af|dvt-bottomLabel
- af|dvt-gaugeBackground
- af|dvt-gaugeFrame
- af|dvt-gaugeSetBackground
- af|dvt-indicator
- af|dvt-indicatorBar
- af|dvt-indicatorBase
- af|dvt-gaugeLegendArea
- af|dvt-gaugeLegendText
- af|dvt-gaugeLegendTitle
- af|dvt-lowerLabelFrame
- af|dvt-metricLabel
- af|dvt-gaugePlotArea
- af|dvt-threshold
- af|dvt-tickLabel
- af|dvt-tickMark
- af|dvt-topLabel
- af|dvt-upperLabelFrame

Before You Begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "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 top labels in a `mySkin.css` file as follows:

   ```
   af|dvt-topLabel
   {
     -tr-font-family:SansSerif;
   }
   ```

   For help with creating a custom skin, see the "Creating an ADF Skin File" section in *Creating ADF Skins with Oracle ADF Skin Editor*.

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 *Creating ADF Skins with Oracle ADF Skin Editor*.

For additional information about using styles and skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

24.5 Formatting Numeric Data Values in Gauges

Gauge child components including `metricLabel`, `tickLabel`, and `gaugeLegendText` display numeric data values in gauges. Each component has a `numberType` attribute that lets you specify whether you want to display the value itself or a percentage that the value represents. In some cases, this might be sufficient numeric formatting.

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.

24.5.1 How to Format Numeric Data Values in Gauges

The metrics represented in gauges are numeric data values. You can apply specific formatting rules to these values.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

> **Note:** When you add a gauge to your page, gauge child components that display configurable data values are automatically added to the `gauge` component, based on the choices you make in the Create Gauge dialog.

To format numeric data values in a gauge:

1. In the Structure window, select the gauge child component displaying data values (**dvt:metricLabel**, dvt:**tickLabel**, or **dvt:gaugeLegendText**) that you wish to configure and choose **Go to Properties**.

   If the component is not available, right-click the **dvt:gauge** component, and choose **Insert Inside Gauge** > **ADF Data Visualization** > (**Metric Label**, **Tick Label**, or **Legend Text**).

2. In the Properties window, if you want to display the data value as a percentage rather than as a value, set the **NumberType** attribute of the component to `NT_PERCENT`.

3. 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:metricLabel** or **dvt:tickLabel**, and choose **Insert Inside** (**Metric Label** or **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.

Note: When the `numberType` attribute of metric or tick labels is set to percent (`NT_PERCENT`), a child `af:convertNumber` tag, if used, will be automatically set to `percent` for its `type` attribute. When `af:convertNumber` is forced to `percent`, gauge clears the `pattern` attribute. This means that patterns are ignored when a gauge forces percent formatting.

24.5.2 What You May Need to Know About Automatic Scaling and Precision

In order to achieve a compact and clean display, gauges automatically determine the scale and precision of the values being displayed in metric labels and tick labels. For example, a value of 40,000 will be formatted as 40K, and 0.230546 will be displayed with 2 decimal points as 0.23.

Automatic formatting still occurs when `af:convertNumber` is specified. Gauge tags that support `af:convertNumber` child tags have `scaling` and `autoPrecision` attributes that can be used to control the gauge's automatic number formatting. By default, these attribute values are set to `scaling="auto"` and `autoPrecision="on"`. Fraction digit settings specified in `af:convertNumber`, such as `minFractionDigits`, `maxFractionDigits`, or `pattern`, are ignored unless `autoPrecision` is set to `off`.

24.6 Adding Gauge Special Effects and Animation

You can add special features to a gauge such as applying gradient effects to parts of a gauge, adding interactivity to gauges, animating gauges, and taking advantage of gauge support for active data.

24.6.1 How to Add Gradient Special Effects to a Gauge

A *gradient* is a special effect in which an object changes color gradually. Each color in a gradient is represented by a stop. The first stop is stop 0, the second is stop 1, and so on. You must specify the number of stops in the special effects for a child component of a gauge that supports special effects.

You can define gradient special effects for the following child components of a gauge:

- `gaugeBackground`
- `gaugeSetBackground`
- `gaugePlotArea`
- `gaugeFrame`
- `gaugeLegendArea`
- `lowerLabelFrame`

- `upperLabelFrame`
- `indicator`
- `indicatorBar`
- `indicatorBase`
- `threshold`

For each child component of a gauge to which you want to add special effects, you must insert a child `specialEffects` component. For example, if you want to add a gradient to the background of a gauge, then you would add a child `specialEffects` component to the `background` component. You must also set the `specialEffects` component `fillType` attribute to `FT_GRADIENT`.

Then, optionally if you want to control the rate of change for the fill color of the child component, you add as many `gradientStopStyle` components as you need to control the color and rate of change for the fill color of the component. The `gradientStopStyle` components are added as child components to the `specialEffects` component.

The approach that you use to define gradient special effects is identical for each child component of the gauge that supports these effects. The procedure defines how to add gradient special effects to the gauge.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

Note: When you add a gauge to your page, gauge child components are automatically added to the `gauge` component based on the gauge type you chose in the Create Gauge dialog.

To add a gradient special effect to a gauge:

1. In the Structure window, select the node that supports gradient special effects (**dvt:gaugeBackground**, for example), and expand if needed to display the child nodes.
2. If the selected node's children do not include a **dvt:specialEffects** component, right-click the node and choose **Insert Inside *node* > Special Effects**.
3. Right-click the **dvt:specialEffects** node and choose **Go to Properties**.
4. In the Properties window, set the following attributes:
   - **FillType**: From the dropdown list select `FT_GRADIENT`.
   - **GradientDirection**: From the dropdown list select the direction of change that you want to use for the gradient fill. The default value is `GD_RIGHT`.
   - **NumStops**: Enter the number of stops to use for the gradient.

5. Optionally, in the Properties window, click **Configure Gradient Stops** to control the color and rate of change for the first gradient stop.
6. In the Properties window, set the following attributes:
   - **StopIndex**: Enter a zero-based integer as an index for the component.
   - **GradientStopColor**: Specify a RGB value for the color that you want to use at this specific point along the gradient. You can also change the color from opaque to transparent. For more information, see Section 24.4.2, "Specifying Transparency for Gauge Elements."
   - **GradientStopPosition**: Enter the proportional distance along a gradient for the identified stop color. The gradient is scaled from 0 to 100. If 0 or 100 is not specified, default positions are used for those points.
7. If you wish to configure additional gradient stops, in the Structure window, right-click the **dvt:specialEffects** component and choose **Insert Inside Special Effects** > **Gradient Stop Style**.
8. Repeat Step 5 through Step 7 for each gradient stop you want to configure.

24.6.2 What Happens When You Add a Gradient Special Effect to a Gauge

When you add a gradient fill to the background of a gauge, specify two stops, and configure the color and rate of change for each stop, XML code is generated. Example 24–6 shows the XML code that is generated.

Example 24–6 XML Code Generated for Adding a Gradient to the Background of a Gauge

```
<dvt:gauge>
  <dvt:gaugeBackground borderColor="#848284">
    <dvt:specialEffects fillType="FT_GRADIENT" gradientDirection="GD_RADIAL">
      <dvt:gradientStopStyle stopIndex="0" gradientStopPosition="60"
           gradientStopColor="FFFFCC"/>
      <dvt:gradientStopStyle stopIndex="1" gradientStopPosition="90"
           gradientStopColor="FFFF99"/>
    </dvt:specialEffects>
  </dvt:gaugeBackground>
</dvt:gauge>
```

24.6.3 How to Add Interactivity to Gauges

Interactivity in gauges involves associating a specified part of a gauge with an HTML attribute such as a hyperlink, or a JavaScript event such as a user moving the cursor over that part of the gauge. For example, a gauge indicator could be associated with a hyperlink, or a tooltip of a gauge indicator could change from "Indicator" to "Indicator is Clicked" when the user clicks the indicator.

You specify interactivity properties on one or more `shapeAttributes` components wrapped in a gauge child `shapeAttributesSet` component. The interactivity provides a connection between the gauge subcomponent, as specified in the `component` attribute of a `shapeAttributes` component, and an HTML attribute or a JavaScript event. Each `shapeAttributes` component must contain a subcomponent and at least one attribute in order to be functional.

The valid values for gauge subcomponents, as specified in the `component` attribute of the `shapeAttributes` component, are:

- `GAUGE_BOTTOMLABEL`: the label below the gauge

- GAUGE_INDICATOR: the indicator in the gauge
- GAUGE_LEGENDAREA: the legend area of the gauge
- GAUGE_LEGENDTEXT: the text label of the legend area
- GAUGE_METRICLABEL: the label showing the metric value
- GAUGE_TOPLABEL: the label above the gauge
- GAUGE_PLOTAREA: the area inside the gauge
- GAUGE_THRESHOLD: the threshold area of the gauge

Interactivity attributes associated with the gauge subcomponent can be any of the following:

- Behavior attributes: An attribute such as onClick, onMouseMove, onKeyDown, or any attribute with a prefix of on that takes a string containing JavaScript code or a reference to a managed bean method that returns JavaScript code as its value. If the value is a managed bean method, the method takes the subcomponent handle as its input parameter.
- Common attributes: An HTML attribute such as alt, href, nohref, target, title, and tabindix that takes a string or a managed bean method that returns a string as its value. The value can be a string or a boolean depending on the attribute. Other attributes control the basic settings of the interactivity, such as clickable, clickAction, and clickListener to control the click events, and id to reference the subcomponent.

For example, Example 24–7 shows the code for a dial gauge where the tooltip of the indicator changes from "Indicator" to "Indicator is Clicked" when the user clicks the indicator, and the tooltip for the gauge metric label displays "Metric Label" when the user mouses over that label at runtime.

Example 24–7 Sample Code for Gauge shapeAttributes Component

```
<dvt:gauge>
  <dvt:shapeAttributesSet>
    <dvt:shapeAttributes component="GAUGE_INDICATOR" alt="Indicator"
                         onClick="document.title="onClick";"/>
    <dvt:shapeAttributes component="GAUGE_METRICLABEL" alt="Metric Label"
                         onMouseMove="document.title="onMouseMove";"/>
  </dvt:shapeAttributesSet>
</dvt:gauge>
```

You can also use a managed bean method to return the value of the interactivity attribute. Example 24–8 shows a managed bean sample code.

Example 24–8 Sample Managed Bean Code

```
public String alt(oracle.dss.dataView.ComponentHandle handle) {
  return handle.getName(); }
public String onClick(oracle.dss.dataView.ComponentHandle handle) {
     return ("document.title=\"onClick\";"); }
public String onMouseMove(oracle.dss.dataView.ComponentHandle handle) {
     return ("document.title=\"onMouseMove\";"); }
```

Example 24–9 shows sample code for referencing the managed bean in a shapeAttributes component.

Example 24–9 Gauge shapeAttributes Component Referencing a Managed Bean

```
<dvt:gauge>
  <dvt:shapeAttributesSet>
    <dvt:shapeAttributes component="GAUGE_INDICATOR" alt="#{sampleGauge.alt}"
                         onClick="#{sampleGauge.onClick}"/>
    <dvt:shapeAttributes component="GAUGE_METRICLABEL"
                         alt="#{sampleGauge.alt}"
                         onMouseMove="#{sampleGauge.onMouseMove}"/>
  </dvt:shapeAttributesSet>
</dvt:gauge>
```

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may find it helpful to understand how managed beans are used in JDeveloper. For more information, see Section 3.6, "Creating and Using Managed Beans."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To add interactivity to a gauge subcomponent:

1. In the Structure window, right-click the **dvt:gauge** component, and choose **Insert Inside Gauge** > **ADF Data Visualizations** > **Shape Attributes Set**.
2. Right-click the **dvt:shapeAttributesSet** node, and choose **Insert Inside Shape Attributes Set** > **Shape Attributes**.
3. Right-click the **dvt:shapeAttributes** node and choose **Go to Properties**.
4. In the Properties window, expand the **Common** Section. For the **Component** attribute, use the dropdown list to select the gauge subcomponent to which you are adding interactivity.

   For example, to add interactivity to the gauge's indicator, choose `GAUGE_INDICATOR` from the **Component** dropdown list.
5. Set one or more of the other attributes in this section to specify the interactivity properties for the subcomponent.

   > **Note:** You can use the attribute dropdown menu on the attributes in this section to choose a Method Expression Builder dialog when creating a reference to a managed bean. For some attributes you can also choose **Edit** > **Edit Property** to select an available managed bean from a dropdown list, or choose **New** to create a managed bean using the Create Managed Bean dialog.

6. Expand the **Behavior** section. Use this section to set one or more of these attributes with a prefix of `on` that takes a string containing JavaScript code or a reference to a managed bean method that returns JavaScript code as its value.

7. If you wish to configure additional interactivity effects for a gauge subcomponent, repeat Step 2 through Step 6 for each subcomponent.

24.6.4 Animating Gauges

You can animate gauges (not gauge sets) upon initial display or to show changes in data. Animation effects are specified in the gauge's `animationOnDisplay` and `animationOnDataChange` properties. For example, a dial gauge indicator can change color at initial display or when a data value increases or decreases. Figure 24–15 shows a dial gauge with the dial indicator animated to display the data change at each threshold level.

Figure 24–15 Animated Dial Gauge

Animation effects can also be performed using active data. The Active Data Service (ADS) allows you to bind ADF Faces components to an active data source using the ADF Model layer. To allow this, you must configure the components and the bindings so that the components can display the data as it is updated in the source. Alternatively, you can configure the application to poll the data source for changes at prescribed intervals.

24.6.5 How to Specify Animation Effects for Gauges

You can set animation effects for gauges upon initial display, or upon data change associated with partial page rerendering (PPR), or Active Data Service (ADS). For more information about PPR, see Chapter 8, "Rerendering Partial Page Content." For more information about ADS, see Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces components. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To specify animation effects for a gauge:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section if needed. Use the **Animation** subsection to set these attributes:

- **AnimationOnDisplay**: Use to specify the type of initial rendering effect to apply. Valid values are:
  - `NONE` (default): Do not show any initial rendering effect.
  - `AUTO`: Apply an initial rendering effect automatically chosen based on graph or gauge type.
- **AnimationOnDataChange**: Use to specify the type of data change animation to apply. Valid values are:
  - `NONE`: Apply no data change animation effects.
  - `AUTO` (default): Apply Active Data Service (ADS) data change animation events. For more information about ADS, see Section 24.6.6, "How to Configure Gauges to Display Active Data."
  - `ON`: Apply partial page refresh (PPR) data change animation events.Use this setting to configure the application to poll the data source for changes at prescribed intervals.
- **AnimationDuration**: Use to specify the animation duration in milliseconds. The default value is `1000`.

24.6.5.1 What You May Need to Know About Skinning and Gauge Animation Effects

You can also use the ADF Data Visualization Tools Skin Selectors to define the animation effects and style globally across all pages in your application by using a cascading style sheet (CSS) to build a skin. By applying a skin to define the animation, the pages in an application will be smaller and more organized, with a consistent style easily modified by changing the CSS file.

Example 24–10 shows an example of using skinning keys to define animation effects.

Example 24–10 Using Skinning Keys to Define Animation Effects

```
af:dvt-gauge
{
  -tr-animation-duration:500;
  -tr-animation-indicators:auto;
  -tr-animation-on-data-change:on;
  -tr-animation-on-display:auto;
}
```

For additional information about using skinning and styles, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

24.6.6 How to Configure Gauges to Display Active Data

Animation effects using Active Data Service (ADS) can be added to dial and status meter gauge types. ADS allows you to bind ADF Faces components to an active data source using the ADF Model layer. To allow this, you must configure the components and the bindings so that the components can display the data as it is updated in the data source. For more information about ADS and configuring your application, see the "Using the Active Data Service" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

You configure a databound gauge to display active data by setting a value on the binding element in the corresponding page definition file.

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces components. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should have a data source that publishes events when data is changed, and you should have created business services that react to those events and the associated data controls to represent those services. For additional information, see the "Using the Active Data Service" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To configure a databound gauge to display active data:

1. In the Structure window, right-click the **dvt:gauge** component, and choose **Go to Page Definition**.
2. In the Structure window, expand **Bindings**, and select the node that represents the attribute binding for the component.
3. In the Properties window, expand the **Advanced** section, and from the **ChangeEventPolicy** attribute dropdown list, select `Push`.

24.7 Using Custom Shapes for Gauges

A set of prebuilt custom shapes styles are provided for the gauge component. You can also create and use a graphics file to create a custom shape for a gauge. Set the `customShapesPath` attribute for the gauge component to use an available custom shapes style or to point to the vector graphics file that is processed into the graphics used for output.

24.7.1 How to Use Prebuilt Custom Shapes Styles

You can choose from a set of prebuilt custom shapes styles to specify a custom shape for a gauge. The custom shapes styles are:

- Rounded rectangle
- Full circle
- Beveled circle

Figure 24–16 shows a dial gauge displayed with each of the custom shapes styles applied.

Figure 24–16 Dial Gauges with Custom Shapes Styles

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To apply a custom shapes style to a gauge:

1. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section, and select the custom shapes style from the **CustomShapesPath** attribute dropdown list. Valid values include: `Rounded Rectangle`, `Full Circle`, and `Beveled Circle`.

24.7.2 How to Use a Custom Shapes Graphic File

Due to the requirements for rotating and resizing a gauge's components, such as the plot area or tick marks, a vector graphics file is required when creating a custom shapes graphic file. Scalable Vector Graphics (SVG) is the supported file format for creating custom shapes for gauges.

After designing the gauge and exporting it to an SVG file, a designer can add information to identify, scale, and position the gauge shapes and components, and to specify other metadata used in processing.

In the SVG file, gauge components are identified using an ID. For example, an SVG file with `<polygon id="indicator"/>` would be interpreted as using a polygon shape for the `indicator` component. To specify multiple shapes to create the desired visual for a component, the ID can be modified as in `id="indicator_0"`, `id="indicator_1"`, and `id="indicator_2"`.

Table 24–1 shows the gauge component IDs and their descriptions.

Table 24–1 Gauge Component IDs for Custom Shapes

ID	Description
indicator	Points to the value represented by the gauge. If not specified, the gauge will use the indicator specified in the application. For the dial gauge, the indicator must be specified while pointing up (90 degrees), so that the shape can be properly rotated. For the status meter gauge, the indicator should be specified with its full extent, and the gauge will be cropped to point to the metric value.
indicatorBase	For a dial gauge, refers to the object that appears at the base of the indicator component. If specified, and the indicatorCenter is not, then the center of the indicatorBase will be taken as the indicatorCenter.
gaugeFrame	Refers to the optional component that adds visual distinction to the plotArea. It can be turned on or off in the application by setting the rendered property. Used primarily when the user wants to use the default gauge plotArea. If no plotArea is specified, then the gauge will insert the default plotArea within the plotAreaBounds. This provides a quick way to change the look of the gauge without having to create a custom plotArea or tickMark.
lowerLabelFrame	Refers to the frame that contains the bottomLabel when its position is LP_BELOW_GAUGE; allows the user to customize the look of this frame. The gauge will position the lowerLabelFrame in the same relative position to other gauge components when it is found in the custom shapes file.
plotArea	For the dial gauge, refers to the circular area within which the indicator moves. For the status meter gauge, refers to the area that contains the indicator. For the LED gauge, refers to the area that contains any graphics that will not be filled with the LED fill color. When a plotArea is not specified, the gauge will draw the default plotArea. For tick marks to be drawn, a specification of the plotArea also requires either tickMarkPath or a set of tick marks.
tickMark	Used to define increments on the gauge. When a set of tick marks is specified with no tickMarkPath, the gauge will use the tick marks exactly where they appear on the plotArea. In this case, it is up to the user to ensure that the tick marks appear at equal increments. If a tickMarkPath is specified, the gauge will accept a single tickMark, at 90 degrees for the dial, and it will rotate and position the tickMark along the tickMarkPath.
upperLabelFrame	Refers to the frame that contains the topLabel when its position is LP_ABOVE_GAUGE. Setting the upperLabelFrame allows the user to customize the look of this frame. The gauge will position the upperLabelFrame in the same relative position to other gauge components when it is found in the custom shapes file.

Table 24–2 shows the metadata IDs and the descriptions used for internal calculations, not rendered in the gauge.

Table 24–2 Metadata IDs for Custom Shapes

ID	Description
`indicatorBarBounds`	Specifies the box containing the minimum and maximum extent of the indicator bar. If not specified, the bounding box is taken to be the entire indicator as specified in the input file.
`indicatorCenter`	Specifies the center of rotation for the indicator that rotates around in a dial gauge. The center of the shape with this ID is considered to be the indicator center. If not specified, it is assumed to be the center of the bottom edge of the plot area for an 180-degree dial gauge, and the center of the plot area for an N-degree dial gauge.
`ledFillArea`	Specifies the area of the LED gauge that should be filled with the appropriate threshold color. If not specified, then the entire `plotArea` shape specified in the graphics file will be filled with the threshold color.
`lowerLabelFrameTextBox`	For complex `lowerLabelFrame` shapes, specifies a rectangle that can be set as the `lowerLabelFrameTextBox`. This box determines the position of the bottom label within the `lowerLabelFrame`.
`plotAreaBounds`	Specified the bounding box for the `plotArea`. If no `plotArea` has been specified in this file, then a bounding box is needed for the gauge to draw the plot area of the gauge. If not specified, then the `gaugeFrame` will use its own bounding box for this purpose.
`thresholdFillArea`	Defines the area that will be filled with the threshold colors. For a dial gauge, specifies the `thresholdFillArea` that will be filled by sweeping an arc from the `indicatorCenter`. For a status meter gauge, specifies the `thresholdFillArea` that will be filled based on the orientation of the status meter gauge.
`tickMarkPath`	Defines the path in which to draw tick marks. This is necessary for the gauge to calculate where tick marks should be drawn on a custom plot area, and the gauge will be unable to change the `majorTickCount` if this is not specified.
`upperLabelFrameTextBox`	For complex `upperLabelFrame` shapes, specifies a rectangle that can be set as the `upperLabelFrameTextBox`. This box determines the position of the `topLabel` within the `upperLabelFrame`.

Example 24–11 shows a sample SVG file used to specify custom shapes for the components of a gauge.

Example 24–11 Sample SVG File Used for Gauge Custom Shapes

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns:svg="http://www.w3.org/2000/svg"
   xmlns="http://www.w3.org/2000/svg"
   version="1.0">

    <rect width="264.72726" height="179.18887" rx="8.2879562"
       ry="10.368411" x="152.76225" y="202.13995"
       style="fill:#c83737;fill-opacity:1;stroke:none"
       id="gaugeFrame"/>
    <rect width="263.09058" height="42.581127" rx="3.0565372"
       ry="3.414634" x="155.11697" y="392.35468"
       fill="#c83737"
```

```
        id="lowerLabelFrame" />
    <rect width="241.79999" height="120.13961"
        x="164.2415" y="215.94714"
        style="fill:#ffeeaa"
        id="plotAreaBounds"/>
    <rect width="74.516975" height="44.101883"
        rx="2.6630435" ry="3.5365853"
        x="247.883" y="325.4415"
        style="fill:#ffd5d5;fill-opacity:1;stroke:none"
        id="indicatorBase"/>
    <rect width="6.0830183" height="98.849045" rx="2.6630435"
        ry="2.2987804" x="282.86035" y="237.23772"
        style="fill:#00aa00;fill-opacity:1;stroke:none"
        id="indicator"/>
</svg>
```

Before you begin:

It may be helpful to have an understanding of how gauge attributes and gauge child tags can affect functionality. For more information, see Section 24.2.1, "Configuring Gauges."

You may also find it helpful to understand additional functionality that can be added using other ADF Faces features. For more information, see Section 24.1.3, "Additional Functionality of Gauge Components."

You should already have a gauge on your page. If you do not, follow the instructions in this chapter to create a gauge. For information, see Section 24.2.2, "How to Add a Gauge to a Page."

To use an existing custom shapes graphics file for a gauge:

1. Import the SVG file into your application and make a note of its location.

   For help with importing files, see the "Managing Applications and Projects" section in *Developing Applications with Oracle JDeveloper*.

2. In the Structure window, right-click the **dvt:gauge** component and choose **Go to Properties**.

3. In the Properties window, expand the **Appearance** section.

4. From the **CustomShapesPath** attribute dropdown menu, choose **Expression Builder** and enter the relative path to the SVG file to be used to specify the custom shapes for your gauge.

   For example, if you imported your file into an `Images` directory under the `Web Content` directory in your project structure, enter the following path in the **CustomShapesPath** field:

   ```
   Images/customShapesFile.svg
   ```

24.7.3 What You May Need to Know About Supported SVG Features

The custom shapes available to you support the following SVG features:

- Transformations
- Paths
- Basic shapes
- Fill and stroke painting

- Linear and radial gradients

SVG features that are not supported by custom shapes include:

- Unit Identifiers: All coordinates and lengths should be specified without the unit identifiers, and are assumed to be in pixels. The parser does not support unit identifiers, because the size of certain units can vary based on the display used. For example, an inch may correspond to different numbers of pixels on different displays. The only exceptions to this are gradient coordinates, which can be specified as percentages.
- Text: All text on the gauge is considered data, and should be specified through the tags or data binding.
- Specifying Paint: The supported options are `none`, 6-digit hexadecimal, and a `<uri>` reference to a gradient.
- Fill Properties: The `fill-rule` attribute is not supported.
- Stroke Properties: The `stroke-linecap`, `stroke-linejoin`, `stroke-miterlimit`, `stroke-disarray`, and `stroke-opacity` attributes are not supported.
- Linear Gradients and Radial Gradients: The `gradientUnits`, `gradientTransform`, `spreadMethod`, and `xlink:href` are not supported. Additionally, the `r`, `fx`, and `fy` attributes on the radial gradient are not supported.
- Elliptical Arc Out-of-Range Parameters: If `rx`, `ry`, and `x-axis-rot` are too small such that there is no solution, the ellipse should be scaled uniformly until there is exactly one solution. The SVG parser will not support this.
- General Error Conditions: The SVG input is expected to be well formed and without errors. The SVG parser will not perform any error checking or error recovery for incorrectly formed files, and it will stop parsing when it encounters an error in the file.

25

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:

25.1 About the Pivot Table Component

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.

25.1.1 Pivot Table and Pivot Filter Bar Component Use Cases and Examples

A pivot table display a grid of data with rows and columns. Figure 25–1 shows a pivot table with multiple attributes nested on its rows and columns.

Figure 25–1 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

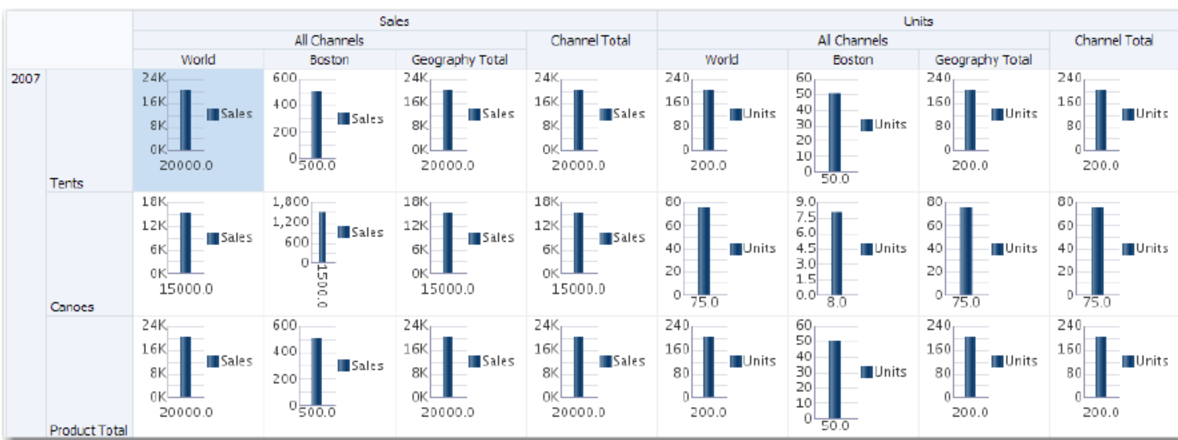
Pivot table data cells support other data display components such as sparkcharts, gauges, and graphs. Figure 25–2 shows a pivot table with sparkcharts illustrating data trends over time in a data cell.

Figure 25–2 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 25–3 shows a pivot table with graphs stamped in data cells.

Figure 25–3 Pivot Table with Graphs Stamped in Data Cells



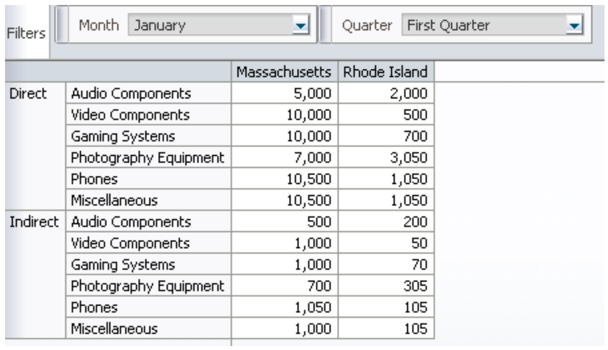
Header and data cells in pivot tables can be customized to display image, icons or links, and to display stoplight and conditional formatting. Figure 25–4 shows a pivot table with conditional formatting to display levels of sales performance.

Figure 25–4 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 25–5 shows a pivot filter bar associated with a pivot table.

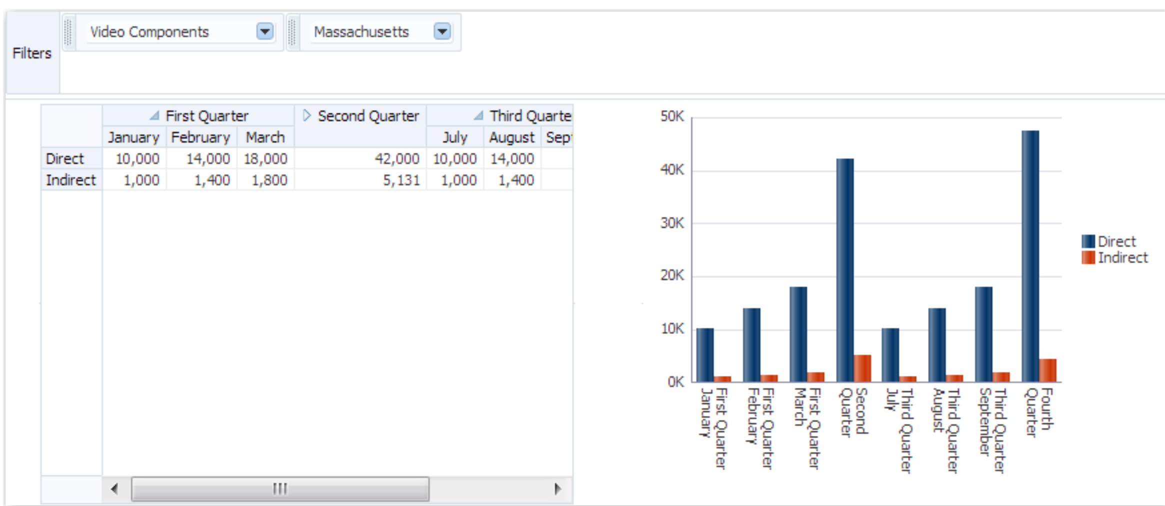
Figure 25–5 Pivot Filter Bar Component Associated with Pivot Table



		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 25–6 shows a filtered view of quarterly sales data displayed simultaneously in a pivot table and on a graph.

Figure 25–6 Pivot Filter Bar Associated with Pivot Table and Graph



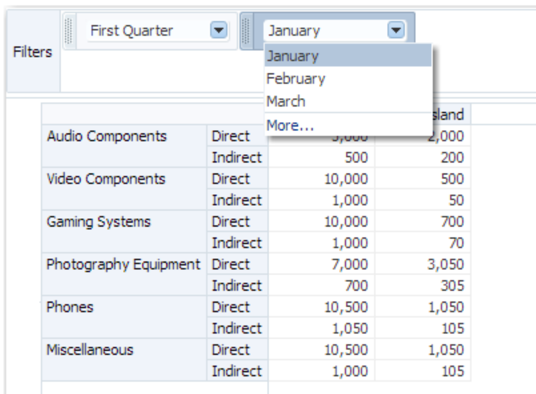
25.1.2 End User and Presentation Features of Pivot Table Components

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.

25.1.2.1 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 25–31 shows a pivot filter bar with Quarter and Month layers that can be used to filter the data displayed in the pivot table.

Figure 25–7 Pivot Filter Bar with Data Layer Filters



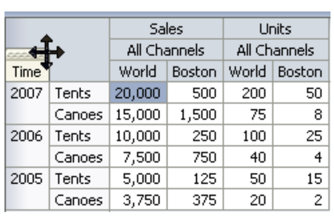
Audio Components	Direct		
	Indirect	500	200
Video Components	Direct	10,000	500
	Indirect	1,000	50
Gaming Systems	Direct	10,000	700
	Indirect	1,000	70
Photography Equipment	Direct	7,000	3,050
	Indirect	700	305
Phones	Direct	10,500	1,050
	Indirect	1,050	105
Miscellaneous	Direct	10,500	1,050
	Indirect	1,000	105

25.1.2.2 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 25–8.

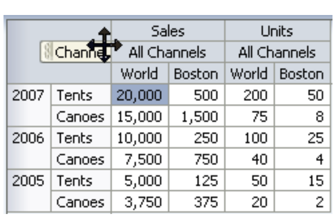
Figure 25–8 Display of Pivot Handle on the Row Edge



		Sales		Units	
		All Channels		All Channels	
Time		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

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 25–9.

Figure 25–9 Display of Pivot Handle on the Column Edge



		Sales		Units	
	Channel	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

If, in Figure 25–8, 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 25–10.

Figure 25–10 Sales Pivot Table After Pivot of Year

	Sales						Units					
	2007		2006		2005		2007		2006		2005	
	All Channels		All Channels		All Channels		All Channels		All Channels		All Channels	
	World	Boston	World	Boston	World	Boston	World	Boston	World	Boston	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.

25.1.2.3 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 25–11 shows a pivot table data cell open for direct editing.

Figure 25–11 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	[illegible]	[illegible]	[illegible]	[illegible]	ain-link	L	1/1/2000
			Boston	500.0	50.0	[illegible]	[illegible]	[illegible]	[illegible]	ain-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

Enter between 2 and 6 characters.

Data cells selected for dropdown list editing are displayed as shown in Figure 25–12.

Figure 25–12 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 coffee milk	42.0	Main-link	S	4/4/2004
2008	Tents	All Channels	World	30000.0	300.0	✓ Item Available	36.0		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.

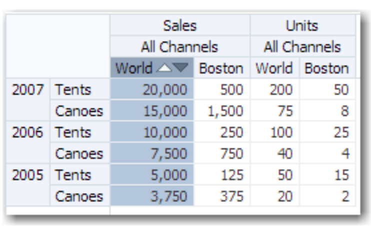
Once editing mode is initiated, users can navigate through read-only data cells to editable data cells, maintaining the editing mode. While an editable cell is selected, you can select other cells using Ctrl or Shift+click without enabling editing in the new cells and maintaining editing in the original cell.

Note: 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.

25.1.2.4 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 25–13 shows the data in the World Sales column sorted descending, where the products within each year are grouped and thereby also sorted descending.

Figure 25–13 Ascending and Descending Sorting Icons in a Pivot Table



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 25–14 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 25–14 Pivot Table Column Header Sorting

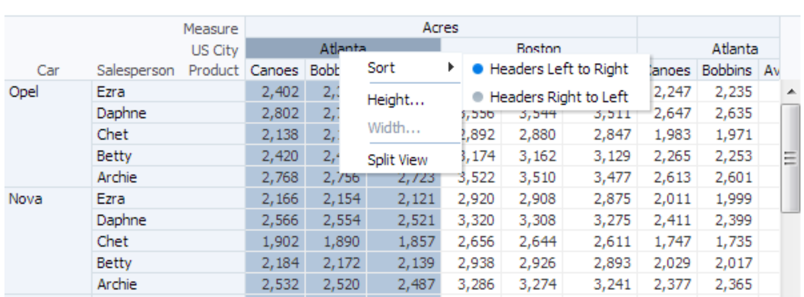
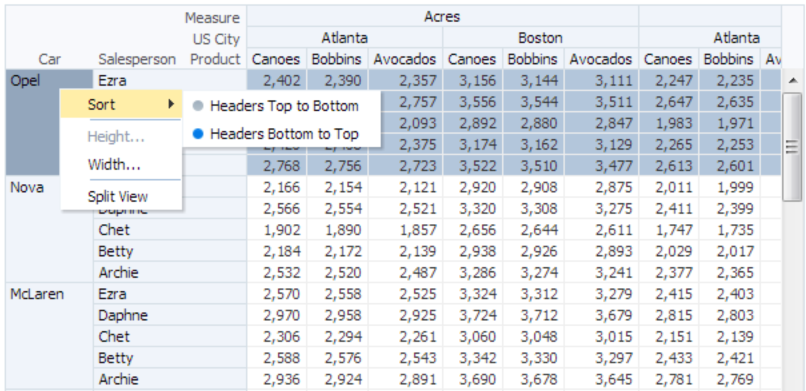


Figure 25–15 shows a pivot table with the Car row sorted with headers from Bottom to Top with a context menu option or change the sort to Top to Bottom.

Figure 25–15 Pivot Table Row Header Sorting



25.1.2.5 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 25–16 and Figure 25–17 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 25–16 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 25–16 Pivot Table with Insert Drilling Enabled

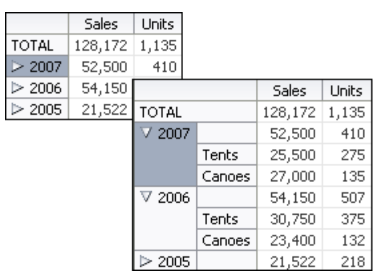
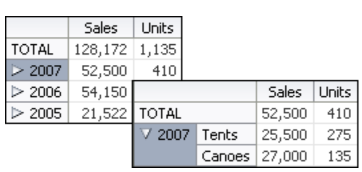


Figure 25–17 shows a pivot table using filter drilling with only the detail numbers are displayed. The numbers for other years, and the aggregated total for all years is filtered out.

Figure 25–17 Pivot Table with Filter Drilling Enabled



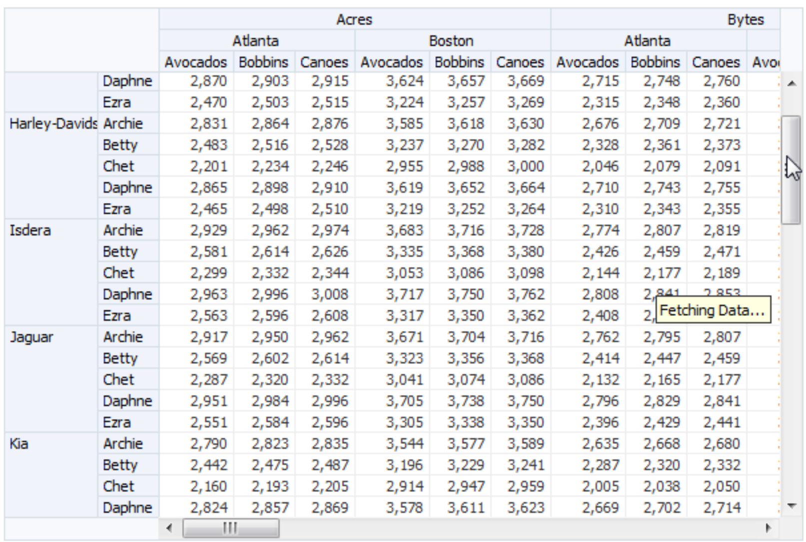
At runtime, a drill icon is enabled in the parent attribute display label for both types of drilling.

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*.

25.1.2.6 Scrolling and Page Controls

Pivot tables support on-demand data scrolling in order to support large data sets while maintaining performance. 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 25–18 shows a pivot table with a large data set using on-demand data scrolling.

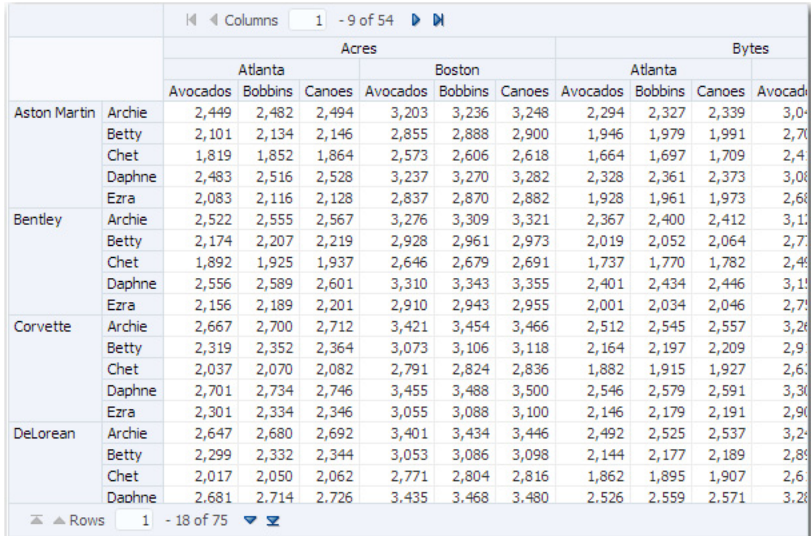
Figure 25–18 On-Demand Data Scrolling in a Pivot Table



		Acres						Bytes			
		Atlanta			Boston			Atlanta			
		Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avocados	Bobbins	Canoes	Avo
	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-Davids	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,[illegible]	Fetching Data...	
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	

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 25–19.

Figure 25–19 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	Avocad
Aston Martin	Archie	2,449	2,482	2,494	3,203	3,236	3,248	2,294	2,327	2,339	3,0
	Betty	2,101	2,134	2,146	2,855	2,888	2,900	1,946	1,979	1,991	2,7
	Chet	1,819	1,852	1,864	2,573	2,606	2,618	1,664	1,697	1,709	2,4
	Daphne	2,483	2,516	2,528	3,237	3,270	3,282	2,328	2,361	2,373	3,0
	Ezra	2,083	2,116	2,128	2,837	2,870	2,882	1,928	1,961	1,973	2,6
Bentley	Archie	2,522	2,555	2,567	3,276	3,309	3,321	2,367	2,400	2,412	3,1
	Betty	2,174	2,207	2,219	2,928	2,961	2,973	2,019	2,052	2,064	2,7
	Chet	1,892	1,925	1,937	2,646	2,679	2,691	1,737	1,770	1,782	2,4
	Daphne	2,556	2,589	2,601	3,310	3,343	3,355	2,401	2,434	2,446	3,1
	Ezra	2,156	2,189	2,201	2,910	2,943	2,955	2,001	2,034	2,046	2,7
Corvette	Archie	2,667	2,700	2,712	3,421	3,454	3,466	2,512	2,545	2,557	3,2
	Betty	2,319	2,352	2,364	3,073	3,106	3,118	2,164	2,197	2,209	2,9
	Chet	2,037	2,070	2,082	2,791	2,824	2,836	1,882	1,915	1,927	2,6
	Daphne	2,701	2,734	2,746	3,455	3,488	3,500	2,546	2,579	2,591	3,3
	Ezra	2,301	2,334	2,346	3,055	3,088	3,100	2,146	2,179	2,191	2,9
DeLorean	Archie	2,647	2,680	2,692	3,401	3,434	3,446	2,492	2,525	2,537	3,2
	Betty	2,299	2,332	2,344	3,053	3,086	3,098	2,144	2,177	2,189	2,8
	Chet	2,017	2,050	2,062	2,771	2,804	2,816	1,862	1,895	1,907	2,6
	Daphne	2,681	2,714	2,726	3,435	3,468	3,480	2,526	2,559	2,571	3,2

Rows 1 - 18 of 75

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.

25.1.2.7 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`.

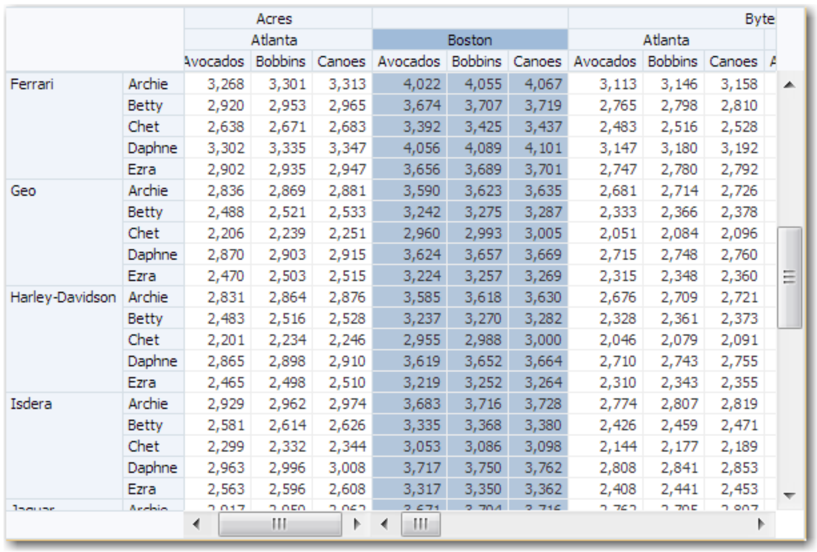
25.1.2.8 Split View of Large Data Sets

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 25–20 shows a columns only split view pivot table data.

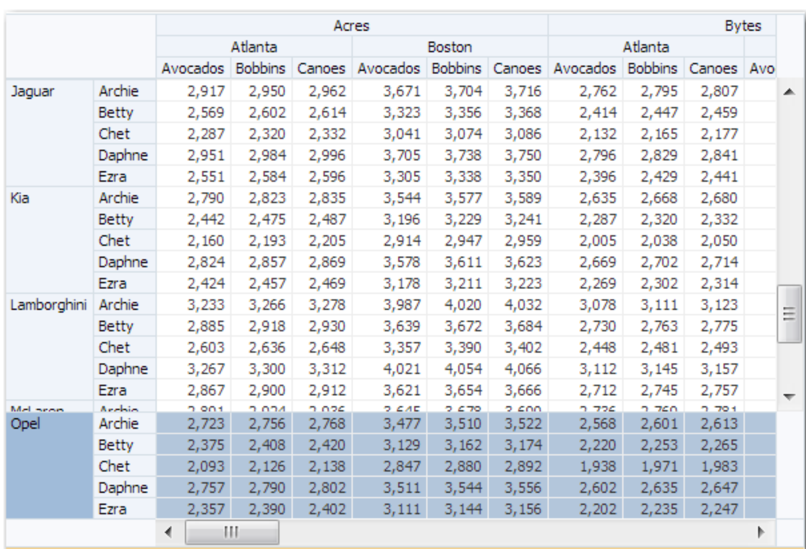
Figure 25–20 Column Only Split View of Pivot Table Data



		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

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 25–21 shows a row only split view of pivot table data.

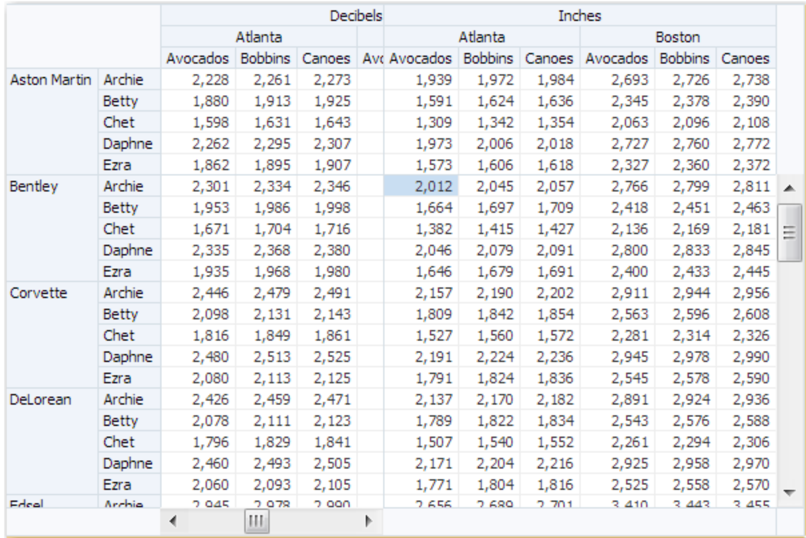
Figure 25–21 Row Only Split View of Pivot Table Data



		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	2,723	2,756	2,768	3,477	3,510	3,522	2,568	2,601	2,613
	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

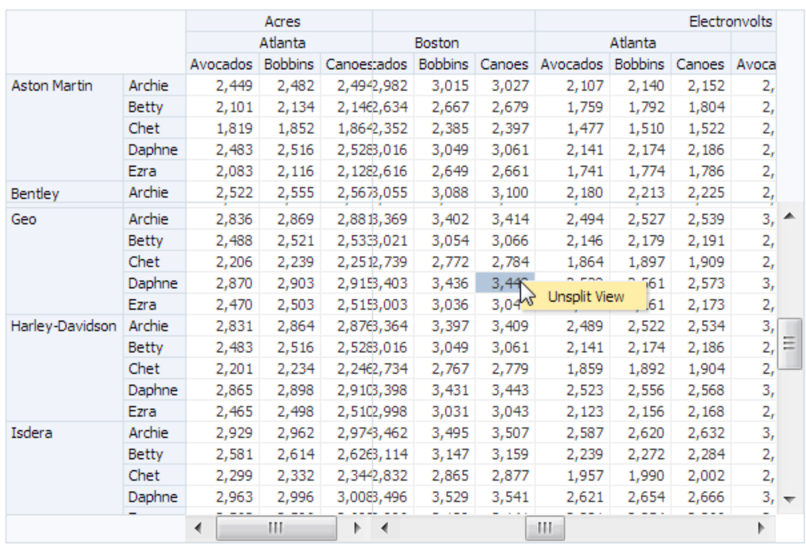
To split both rows and columns, select the data cell that should be the first cell of the last pane, and in the context menu select **Split View**. Figure 25–22 shows a row and column split view of pivot table data.

Figure 25–22 Row and Column Split View of Pivot Table Data



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 Example 25–23.

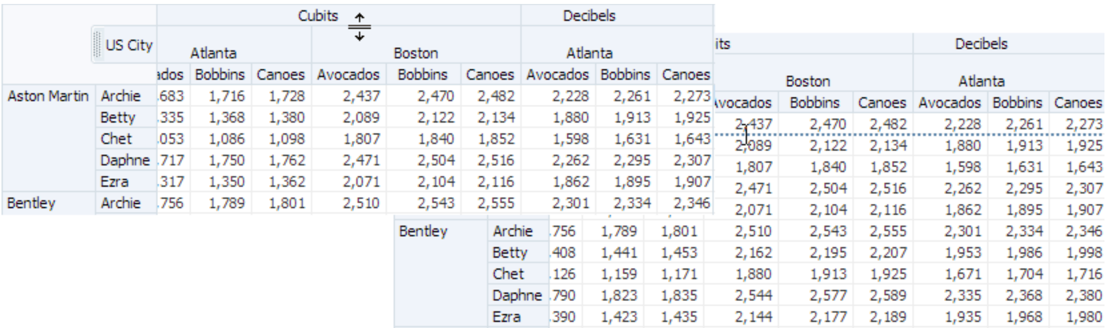
Figure 25–23 Pivot Table Unsplit View Command



25.1.2.9 Sizing

The default size of a pivot table is a width of 300 pixels and a height of 300 pixels. The pivot table autosizes rows, columns, and layers within the space allowed when the pivot table is initially displayed. 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 25–24 shows the double-sided arrow and dotted line resize indicators.

Figure 25–24 Pivot Table Resize Indicators



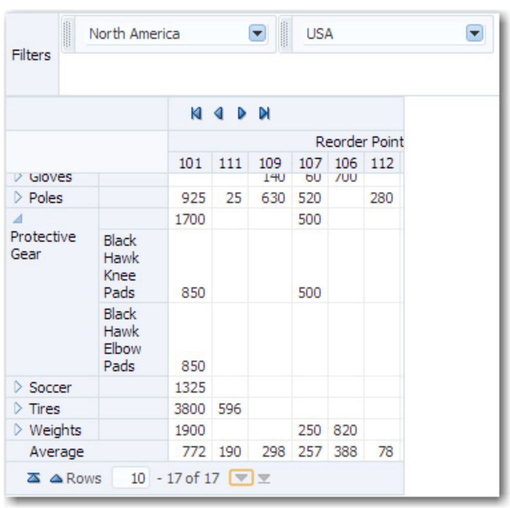
When you resize rows, columns, or layers, the new sizes remain until you perform a pivot operation. After a pivot operation, the new sizes are cleared and the pivot table rows, columns, and layers return to their original sizes.

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.

25.1.2.10 Header Cell Word Wrapping

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 25–10 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 25–25 Pivot Table Header Cell Word Wrapping



25.1.2.11 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` 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 that 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 Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

25.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: By default, pivot table and pivot filter bar components are accessible. You can configure your application pages with pivot table and pivot filter bar components to be accessible to screen reader users. For more information, see Chapter 33, "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 Appendix D, "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 Chapter 31, "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 Section 12.2.2, "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 Section 22.2, "Common Functionality in Data Visualization Components."

25.2 Using the Pivot Table Component

To use the `pivotTable` component, define the data, add the pivot table to a page, and complete the additional configuration in JDeveloper.

25.2.1 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`.

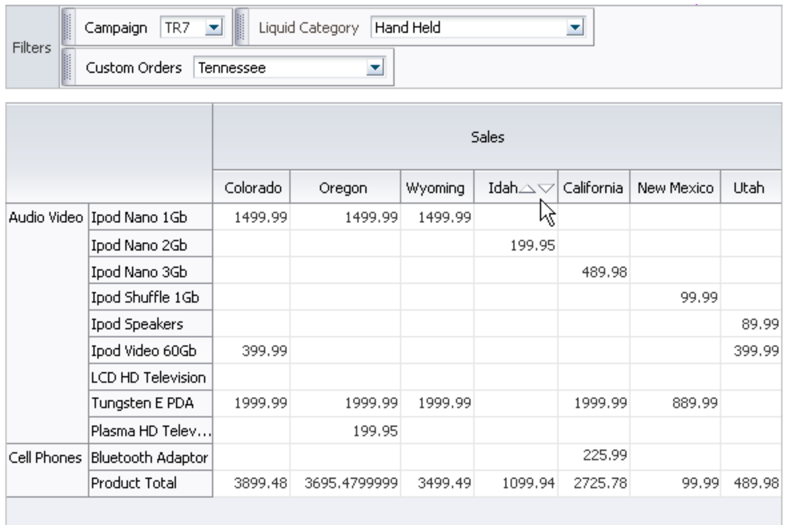
Pivot tables require that the `value` attribute is set in JDeveloper. If you are using UI-first development, the value of the `value` attribute must be stored in the pivot table's data model or in classes and managed beans.

25.2.2 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. 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 25–26 shows a pivot table and its associated pivot filter bar displaying the sales of electronic equipment.

Figure 25–26 Electronic Sales Pivot Table



		Sales						
		Colorado	Oregon	Wyoming	Idah...	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	
	Plasma HD Telev...		199.95					
Cell Phones	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 25–26:

- 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 25–26, 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 25–26, 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 25–26, 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 25–26, 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=Colorado

Likewise, 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 1Gb

Finally, the QDR for the "Sales" header cell contains the following information:

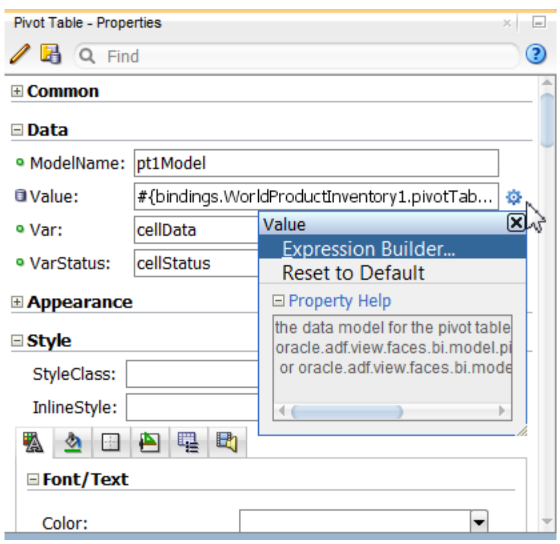
– Measure=Sales

25.2.3 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 25–27 shows the dropdown menu for a pivot table component `value` attribute.

Figure 25–27 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 Section 25.2.2, "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 Section 25.1.3, "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 Section 25.3.1, "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 Section 25.6, "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 Section 22.2.1, "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**.

25.2.4 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. Example 25–1 shows custom CSS styling using `inlineStyle` and `contentStyle` attributes of a data cell `af:outputText` and `af:inputText` respectively.

Example 25–1 Code Sample for Data Cell CSS Styling

```
<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 Chapter 31, "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 Chapter 9, "Organizing Content on Web Pages."

25.2.5 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 Section 25.3, "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
- View a live data preview of the 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*.

25.2.6 What You Many 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 scrollbar 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 25–19. To configure a page control, 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 Section 9.2.1, "Geometry Management and Component Stretching."
- Set the `scrollPolicy` attribute to `auto` (if the page may also run on a desktop device) or `page` (if the page will only run on a mobile device.

If the pivot table is not in a flowing container, or if those attributes are not set correctly, the pivot table will display a scrollbar instead of pages.

25.2.7 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 Appendix D, "Creating Web Applications for Touch Devices Using ADF Faces."

25.3 Configuring Header and Data Cell Stamps

Each immediate child of a 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.

25.3.1 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 25–1 shows the metadata keywords supported for data cells in a rowset data model.

Table 25–1 Supported Metadata Keywords for Data Cells

Keyword	Description
`dataValue`	Most frequently useful keyword. Returns the data value `Object` for the current cell. To specify the object’s accessible field through EL Expression, use the setting `dataValue.`*`fieldName`*.
`dataCubeMax` and `dataCubeMin`	Returns a number that is the maximum and minimum, respectively, for the measure of the cell across the values in the cube.
`dataIsTotal`	Returns a Boolean `true` if this cell is an aggregate.
`dataAggregates`	If the cell is an aggregate, returns a `List<String,Object>` of the column and value pairs representing the cells (nonaggregate) that make up the aggregation for the given cell.
`aggregateCollection`	If the cell is an aggregate, returns the `List<String,Object>` of the column and value pairs in the cube that make up the cell's aggregate value. Note that `aggregateCollection` is post-cube and dataAggregates is not.
`dataRow`	Returns a `Map<String,Object>` from attribute name to data Object in the original row mapping. Usage: dataRow.foo, where "foo" is one of the rowset attribute (column) names.
`dataTypeColumn`	Returns a String representing the name of the rowset attribute from which the value comes.
`dataRowKey`	Returns the row data model's ADF Model row key,
`dataKeyPath`	Returns the ADF Model key path object.

Table 25–2 shows the metadata keywords supported for header cells in a rowset data model.

Table 25–2 Supported Metadata Keywords for Header Cells

Keyword	Description
`dataValue`	Most frequently useful keyword. Returns the data value `Object` for the current cell. To specify the object’s accessible field through EL Expression, use the setting `dataValue.`*`fieldName`*.
`value`	Returns the String value of the header cell. Also available in cubic data models.
`label`	Returns the String label for the header cell. Also available in cubic data models.
`isTotal`	Returns a Boolean `true` 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 `layerName`.

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.

Example 25–2 shows a code sample for using var and varStatus to access data from a stamped data cell. The sample also illustrates using var and varStatus to format the pivot table based on the header cell stamp.

Example 25–2 Code Sample for Using var and varStatus Properties

```
<dvt:pivotTable
                  id="pivotTable3"
                  value="#{pivotTableMemberFormatting.dataModel}"
                  var="cellData"
                  varStatus="cellStatus">
  <dvt:headerCell>
    <af:switcher
                 facetname="O___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.

Figure 25–28 shows the pivot table resulting from the code sample.

Figure 25–28 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 and gauges in pivot tables. Example 25–3 shows a code sample using sparkcharts stamped in data cells. The resulting pivot table is shown in Figure 25–2.

Example 25–3 Code Sample for Stamping Sparkcharts in Data Cells

```
<dvt:pivotTable id="pivotTable1"
                value="#{pivotTableSparkChart.dataModel}"
                var="cellData"
                varStatus="cellStatus">
  <dvt:dataCell>
    <af:switcher id="s2"
                 facetname="O___b_cellData_dataIsTotal__b__"
                 defaultFacet="false">
      <f:facet name="true">
        <dvt:sparkChart id="sc1" shortDesc="Spark Chart"
                        highMarkerColor="#008200"
                        lowMarkerColor="#ff0000">
          <af:iterator id="i1"
                       value="#{cellData.aggregateCollection}"
                       var="sparks" >
            <dvt:sparkItem id="si1"
                           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="O___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>
```

Example 25–4 shows a code sample for using gauges in data cells. The resulting pivot table is displayed in Figure 25–3.

Example 25–4 Code Sample for Stamping Gauges in Data Cells

```
<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>
```

25.3.2 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 Section 9.13, "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 Section 25.1.2.3, "Editing Data Cells."

Example 25–5 shows a code sample for configuring 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. The example also illustrates components that can be used as children of `headerCell`.

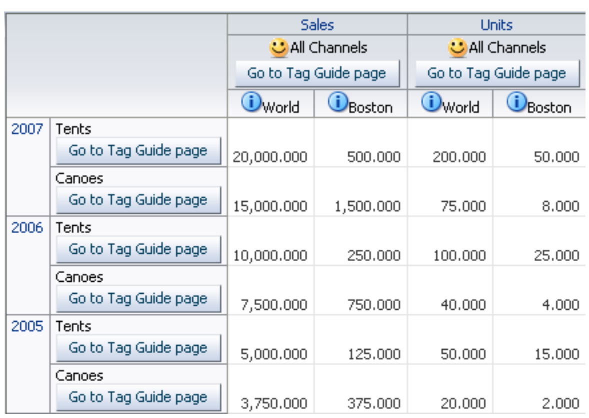
Example 25–5 Code Sample for Header Cell Stamping

```
<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="cil1"/>
            <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 25–29 shows the resulting pivot table for the code sample.

Figure 25–29 Pivot Table Header Cell Stamps



		Sales		Units	
		All Channels / Go to Tag Guide page		All Channels / 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

Example 25–6 shows a code sample for configuring 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. The example also illustrates components that can be used as children of `dataCell`.

Example 25–6 Code Sample for Data Cell Stamping

```
<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" />
        </af:inputText>
      </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>
      <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 25–30 shows the resulting pivot table for the code sample.

Figure 25–30 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 Section 25.2.2, "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 Section 25.2.3, "How to Add a Pivot Table to a Page."

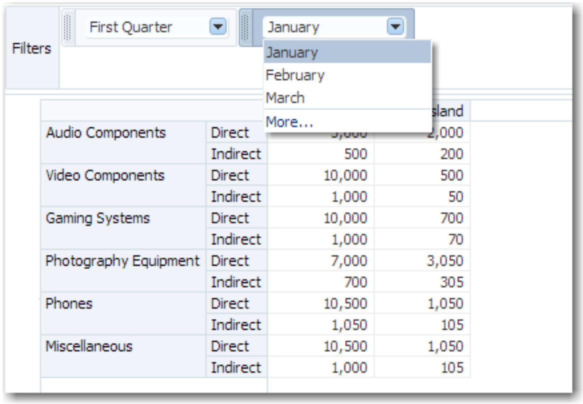
To add and configure a header or data cell stamp:

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.

25.4 Using Pivot Filter Bars

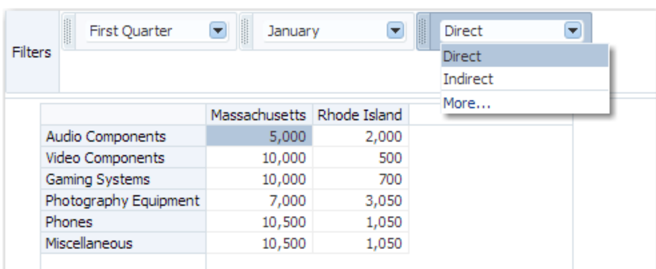
You can enhance the data filtering capacity in a pivot table by adding a pivot filter bar. Zero or more layers of data not already displayed in the pivot table row edge or column edge are displayed in the page edge. Figure 25–31 shows a pivot filter bar with Quarter and Month layers that can be used to filter the data displayed in the pivot table.

Figure 25–31 Pivot Filter Bar with Data Layer Filters



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 25–32 shows the modified pivot table and pivot filter bar when the Channel data layer is pivoted to the page edge.

Figure 25–32 Pivot Table and Pivot Filter Bar After Pivot



You can style pivot filter bars using `inlineStyle` and `styleClass` attributes and skinning keys. For more information, see Section 25.2.4, "Configuring Pivot Table Display Size and Style."

25.4.1 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. Example 25–7 shows a code sample for associating a pivot filter bar with a pivot table.

Example 25–7 Code Sample for Pivot Filter Bar

```
<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.

25.4.2 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 shows a filtered view of quarterly sales data displayed in both a pivot table and on a graph as illustrated in Figure 25–6.

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 Section 4.8, "Locating a Client Component on a Page." For more information about PPR, see Chapter 8, "Rerendering Partial Page Content."

Example 25–8 shows a code sample for using a pivot filter bar with a pivot table and graph as illustrated in Figure 25–6.

Example 25–8 Code Sample for Pivot Filter Bar partialTriggers

```
<dvt:pivotFilterBar id="pfb1" binding="#{editor.component}"
                    value="#{pivotFilterBar.queryDescriptor}" modelName="model1"
                    styleClass="AFStretchWidth"/>
<af:panelGroupLayout layout="horizontal" id="pgl2">
  <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>
```

25.5 Adding Interactivity to Pivot Tables

Pivot tables and pivot filter bars support user operations including selection, exporting to an spreadsheet, and displaying in printable mode.

25.5.1 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.

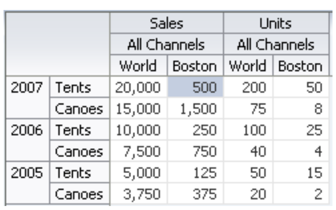
An application can implement features such as displaying customized content for a context menu, based on currently selected cells. Example 25–9 shows sample code for getting the currently selected header cells.

Example 25–9 Sample Code to Get 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 25–11.

Figure 25–33 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

25.5.2 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 Chapter 8.1, "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.

Example 25–10 shows sample code for updating a pivot table by displaying the totals when a checkbox is triggered. The triggering component uses the ID as the `partialTriggers` value.

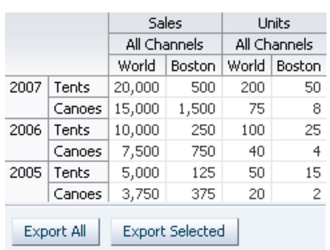
Example 25–10 Partial Update of a Pivot Table

```
<dvt:pivotTable id="goodPT"
    value="#{richPivotTableModel.dataModel}"
    partialTriggers="showTotals"/>

  <af:selectBooleanCheckbox id="showTotals" autoSubmit="true" label="Show Totals"
    value="#{richPivotTableModel.totalsEnabled}"/>
```

25.5.3 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 25–34 shows a pivot table that includes button components that allow users to export the data to an Excel spreadsheet.

Figure 25–34 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

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`. Example 25–11 shows the code sample for the **Export to Excel** buttons.

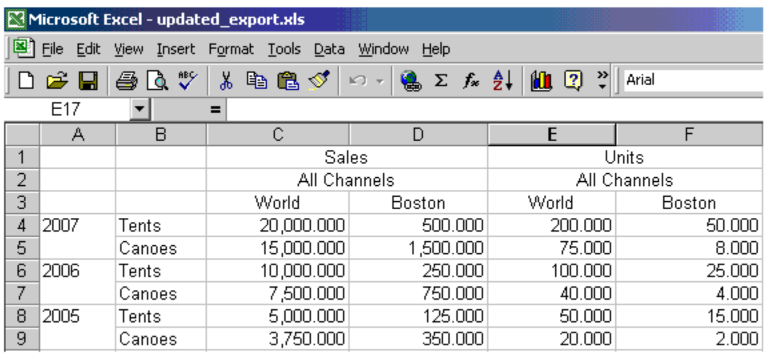
Example 25–11 Code Sample for Export to Excel Button

```
<dvt:pivotTable id="pivotTableToExport"
               binding="#{editor.component}"
               contentDelivery="immediate"
               value="#{pivotTableExport.dataModel}" summary="pivot table"/>

<h:panelGrid id="pfl" 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 25–35 shows the resulting Excel spreadsheet when the Export All button is clicked.

Figure 25–35 Pivot Table Export to Excel Spreadsheet



	A	B	C	D	E	F
1			Sales		Units	
2			All Channels		All Channels	
3			World	Boston	World	Boston
4	2007	Tents	20,000.000	500.000	200.000	50.000
5		Canoes	15,000.000	1,500.000	75.000	8.000
6	2006	Tents	10,000.000	250.000	100.000	25.000
7		Canoes	7,500.000	750.000	40.000	4.000
8	2005	Tents	5,000.000	125.000	50.000	15.000
9		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.

25.5.4 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 Chapter 37, "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.

25.6 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. For more information about using `inlinineStyle` and `styleClass` attributes. For information, see Section 25.2.4, "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.

25.6.1 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.

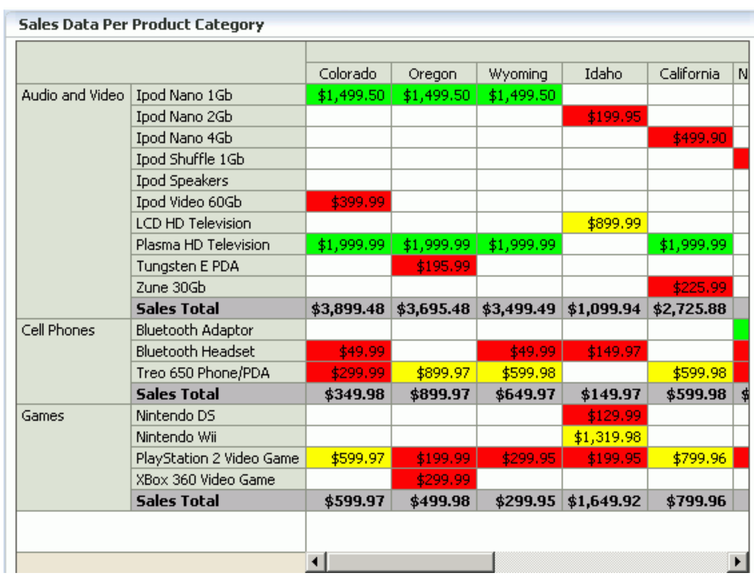
2. Pass the `DataCellContext` to a method expression for the `dataFormat` attribute of the pivot table.
3. In the method expression, write code that specifies the kind of formatting you want to apply to the data cells of the pivot table. This method expression must return a `CellFormat` object.

25.6.2 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 25–36 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 Section 25.6.3, "Configuring Stoplight and Conditional Formatting Using CellFormat."

Figure 25–36 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**	

Example 25–12 shows sample code that produces the required custom formats for the sales totals, but not for the stoplight formatting. 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 Example 25–13.

Example 25–12 Sample Code to Change Style in a Pivot Table

```
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;
  }
```

25.6.3 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 25–36 shows data cells with stoplight formatting for minimum, acceptable, and below standards sales for States.

Example 25–13 shows code that performs stoplight formatting in a pivot table that does not display totals. If you want to do stoplight formatting for a pivot table that displays totals, then you might want to combine the code from Example 25–12 (which addresses rows with totals) with the code for stoplight and conditional formatting.

Example 25–13 Sample Code for Stoplight and Conditional Formatting

```
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 (isStoplightingEnabled())
{
  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;
}
```

26

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:

- Section 26.1, "About the Gantt Chart Components"
- Section 26.2, "Using the Gantt Chart Components"
- Section 26.3, "Customizing Gantt Chart Tasks and Resources"
- Section 26.4, "Customizing Gantt Chart Display Elements"
- Section 26.5, "Adding Interactive Features to Gantt Charts"

26.1 About the Gantt Chart Components

A *Gantt chart* is a type of horizontal bar graph that you use to plan and track projects. It shows resources or tasks in a time frame with a distinct beginning and end. A Gantt chart component is composed of two regions, 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 more information about ADF tree tables, including virtualization of rows, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."

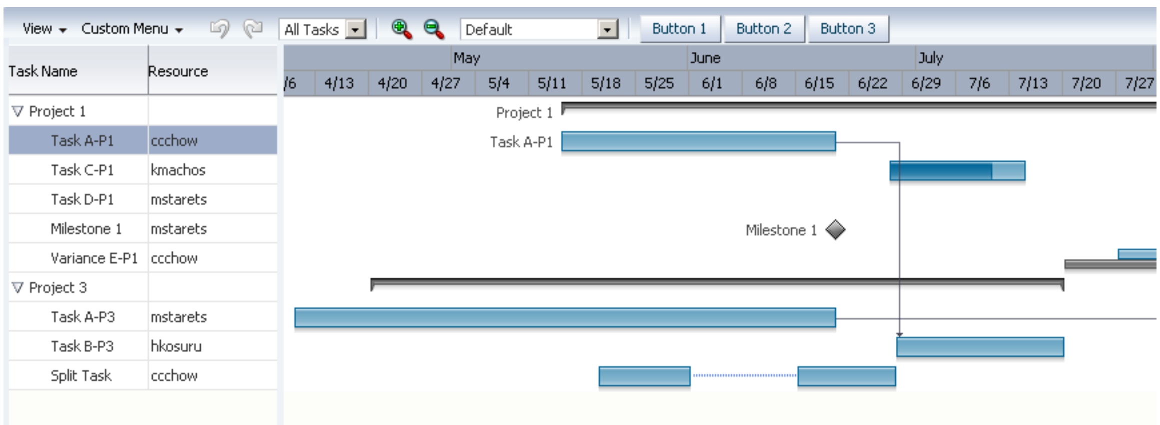
ADF Gantt chart components include a project Gantt chart (`projectGantt`), a resource utilization Gantt chart (`resourceUtlizationGantt`), and a scheduling Gantt chart (`schedulingGantt`).

26.1.1 Gantt Chart Component Use Cases and Examples

The Gantt chart provides the following types:

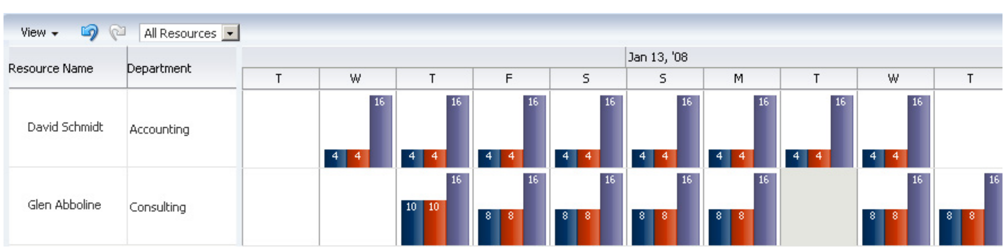
- Project Gantt chart: A project Gantt chart is used for project management. The chart lists tasks vertically and shows the duration of each task as a bar on a horizontal time line. It graphs each task on a separate line as shown in Figure 26–1.

Figure 26–1 Project Gantt Chart for a Software Application



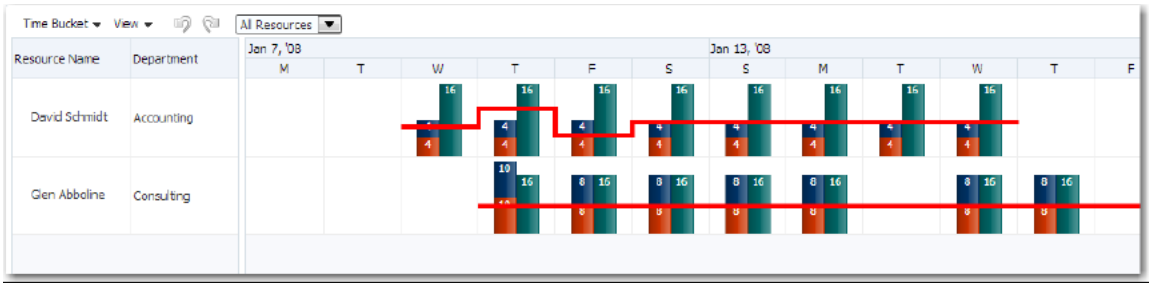
- Resource Utilization Gantt chart: A resource utilization Gantt chart graphically shows the metrics for a resource, for example, whether resources are over or under allocated. It shows resources vertically while showing their metrics, such as allocation and capacity on the horizontal time axis. Figure 26–2 shows a resource utilization Gantt chart illustrating how many hours are allocated and utilized for a particular resource in a given time period.

Figure 26–2 Resource Utilization Gantt Chart for a Software Application



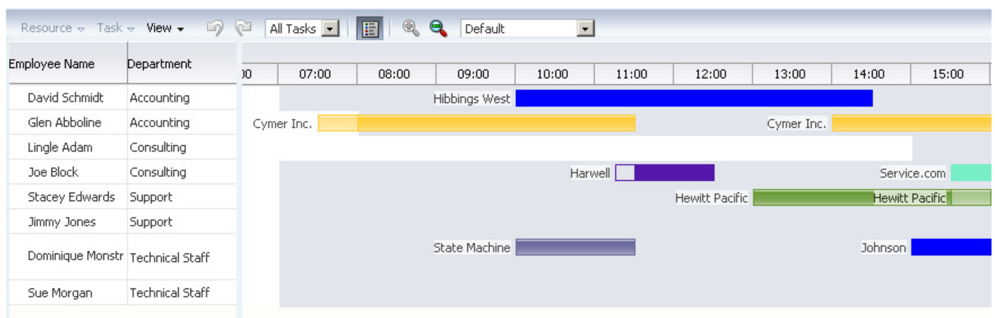
The metrics for a resource utilization Gantt chart can also be configured to display as stacked bars, or as a horizontal line. Figure 26–3 shows a resource utilization Gantt chart illustrating a stacked bar representing hours utilized for two metrics, a vertical bar representing hours allocated, and a horizontal line representing a threshold metric that steps through the chart.

Figure 26–3 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 26–4.

Figure 26–4 Scheduling Gantt Chart for a Software Application



26.1.2 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.

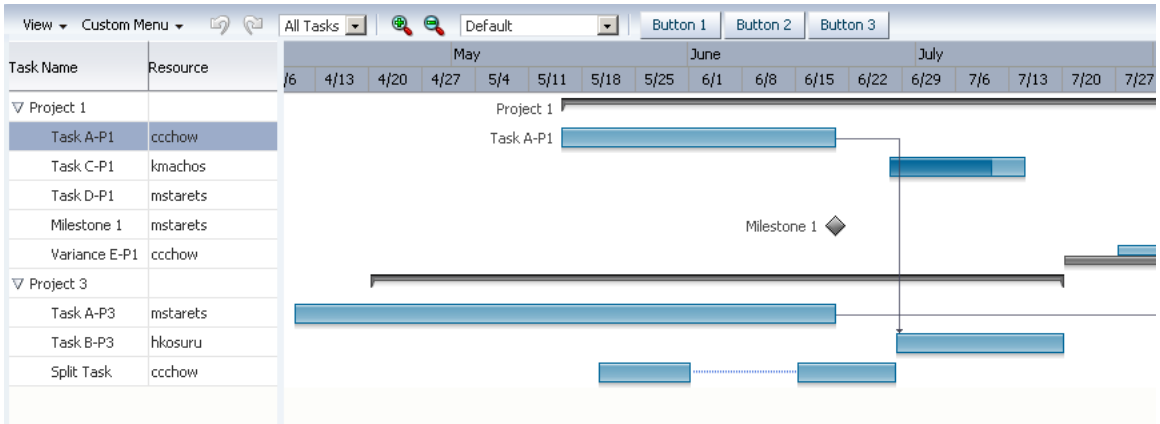
26.1.2.1 Gantt Chart Regions

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 26–4, 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.

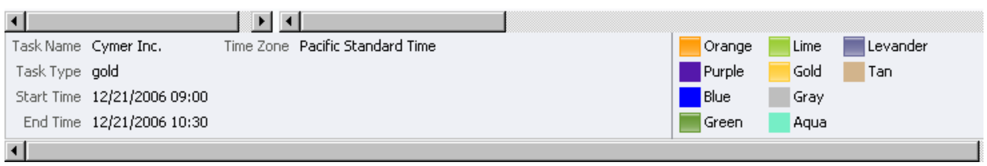
Figure 26–5 Project Gantt Chart Regions



26.1.2.2 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 26–6 shows an information panel for the scheduling Gantt chart in Figure 26–4 with information about a task selected in the chart region and the Gantt chart legend.

Figure 26–6 Scheduling Gantt Chart Information Panel



26.1.2.3 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 26–7 displays the menu bar, which is visible in the Gantt chart by default. You can change the visibility of the menu bar and customize menu items.

Figure 26–7 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.

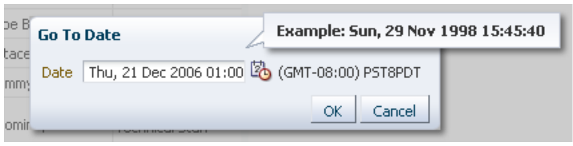
– Expanding and collapsing the display of hierarchical data in the table region.

– Hiding and showing dependency lines between tasks in the chart region.

– Hiding and showing the Gantt chart legend.

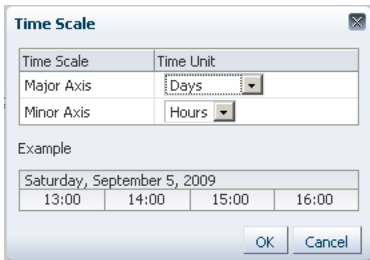
– Specify a specific date in the Gantt chart. Figure 26–8 shows the View menu Go to Date dialog.

Figure 26–8 Go to Date Dialog



– Changing the time scale of the Gantt chart. Figure 26–9 shows the View menu Time Scale dialog.

Figure 26–9 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 26–10 shows a sample toolbar for a project Gantt chart.

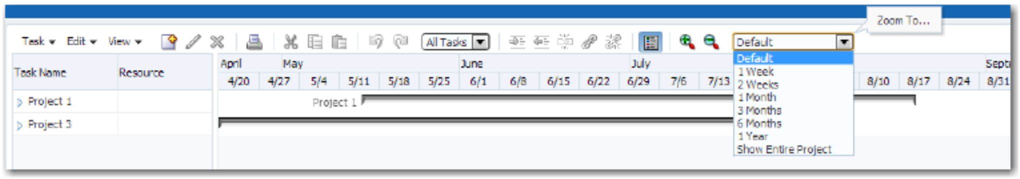
Figure 26–10 Sample Toolbar for a Project Gantt Chart

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 26–11 shows the zoom-to-fit toolbar option expanded for a project Gantt chart.

Figure 26–11 Zoom-to-Fit Toolbar Option

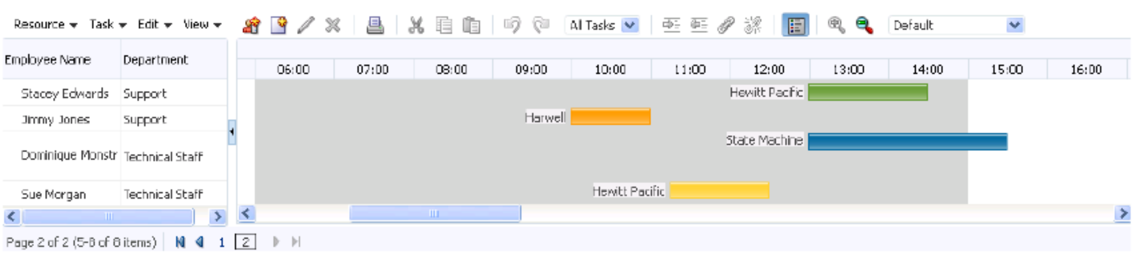


26.1.2.4 Scrolling, Zooming, and Panning

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 26–12 shows a resource utilization Gantt chart configured with a horizontal page control.

Figure 26–12 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.

26.1.2.5 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 26–13 shows a project Gantt chart **View** menu with options for showing dependencies as lines, menu items, or not displayed.

Figure 26–13 Project Gantt Chart Show Dependencies Options

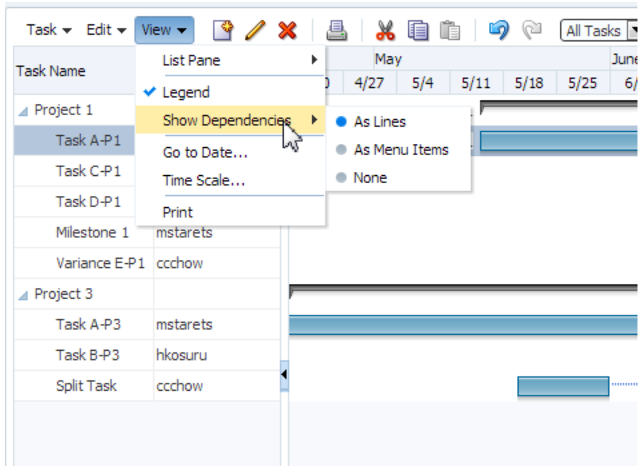
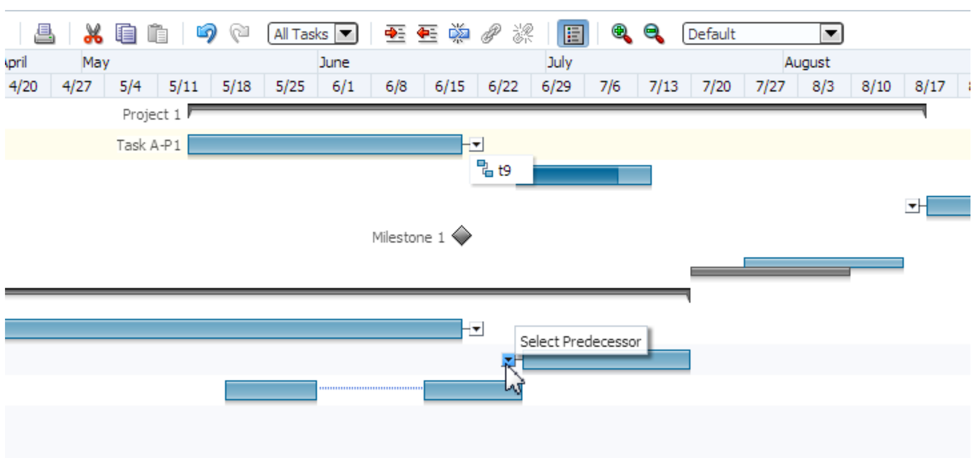


Figure 26–14 shows a project Gantt chart with dependency task menu items for a successor and predecessor task.

Figure 26–14 Project Gantt Chart Dependency Task Menu Items

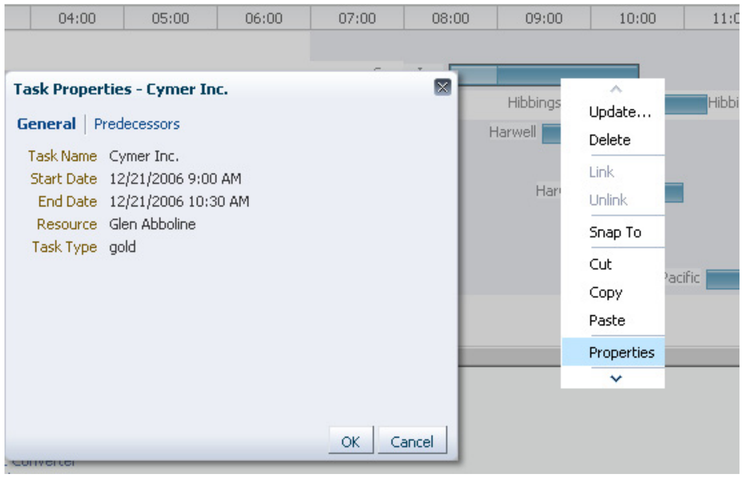


26.1.2.6 Context Menus

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 Section 26.4.6, "Customizing Gantt Chart Context Menus."

Figure 26–15 shows a custom context menu item displayed for a scheduling Gantt chart task properties.

Figure 26–15 Custom Context Menu for Task



26.1.2.7 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 Section 26.5.1, "Performing an Action on Selected Tasks or Resources."

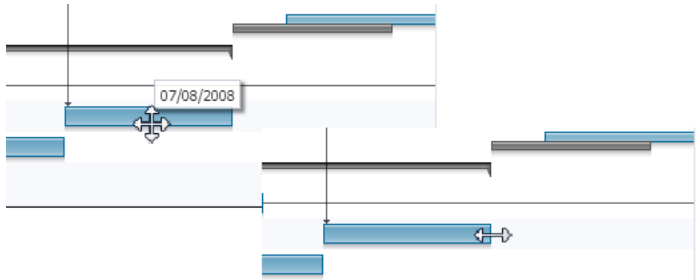
Note: If you configure your component to allow multiple selection, users can select one row and then press the shift key to select another row, and all the rows in between will be selected. This selection will be retained even if the selection is across multiple data fetch blocks. Similarly, you can 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.

26.1.2.8 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 curser. 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 26–16 shows the move and resize cursors for an editable task in a project Gantt chart.

Figure 26–16 Move and Resize Cursors for an Editable Task Bar



26.1.2.9 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.

26.1.2.10 Printing

The Gantt chart provides printing capability in conjunction with Apache or XML Publisher by generating PDF files. For more information, see Section 26.5.4, "Printing a Gantt Chart".

26.1.2.11 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.

26.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: By default, Gantt chart components are accessible. You can make your application pages accessible for screen reader users. For more information, see Chapter 33, "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 Section 12.12, "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 Section 12.2.2, "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 Section 22.2, "Common Functionality in Data Visualization Components."

26.2 Using the Gantt Chart Components

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. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

26.2.1 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 26–1 shows a complete list of data object keys for the project Gantt chart.

Table 26–1 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.

Table 26–1 (Cont.) Data Object Keys for Project Gantt

Data Object Key	Date Type and Description
Dependency (node)	A list of dependencies for a task. Data object keys for dependencies include: ■ 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.
percentComplete	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.
startTime (required)	Date. The starting time for all tasks.
Subtasks (node)	An optional list of subtasks for all tasks.
taskId (required)	String. The unique identifier for all tasks.
type	Sting. The type of the tasks for all tasks.

26.2.2 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 26–2 shows a complete list of data object keys for the resource utilization Gantt chart.

Table 26–2 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 `top` (default) and `inside`.
`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.

26.2.3 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 task.
- `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.

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 26–3 shows a complete list of data object keys for a scheduling Gantt chart.

Table 26–3 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: ■ `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`.
`endTime` (required)	Date. The end time for all the 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 task bar for all tasks.
`icon3`	String. The third icon associated with the task 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`, `inside_left`, `inside_right`, and `inside_center`. In locales using right-to-left display, `start` and `end` values are also supported.
`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`, `inside_left`, `inside_right`, and `inside_center`. In locales using right-to-left display, `start` and `end` values are also supported.
Recurring tasks (node)	A list of recurring tasks for all tasks.
`resourceId` (required)	String. The unique identifier of a resource.
Split tasks (node)	A collection of tasks without a continuous time line for all tasks.
`startTime` (required)	Date. The start time for all tasks.
`startupTime`	Date. The startup time before a task begins.
Tasks (node) (required)	A list of tasks associated with a resource.
`taskId` (required)	String. The unique identifier of the task for all tasks.
`taskType`	String. The type of the task for all tasks.
`workingDaysOfTheWeek`	Object. A list of the working days of the week.
`workingEndTime`	Date. The work end time for the resource.
`workingStartTime`	Date. The work start time for the resource.

26.2.4 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:

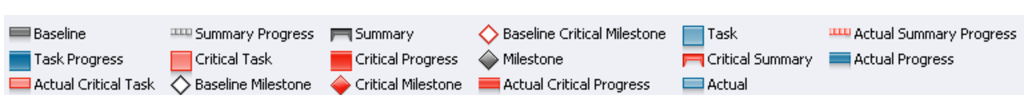
- Normal: The basic task type. It is a plain horizontal bar that shows the start time, end time, and duration of the task.
- Summary: The start and end date for a group of subtasks. A summary task cannot be moved or extended. Instead, it is the responsibility of the application to execute code to recalculate 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 26–17 displays a legend that shows common task types in a project Gantt chart.

Figure 26–17 Project Gantt Chart Legend for Task Types



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.

26.2.5 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. Example 26–1 shows sample code for a project Gantt chart with "Task Name" as the nodeStamp facet, with columns for Resource, Start Date, and End Date.

Example 26–1 Sample Code for Project Gantt Chart Columns

```
<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 26–4 shows the facets supported by Gantt chart components.

Table 26–4 Facets Supported by Gantt Chart Components

chartPopupMenu	Specifies the component to use to identify additional controls to appear in the context menu of the chart region. Must be an `af:popup` component.
customPanel	Specifies the component to use to identify controls to appear in the custom tab of the task properties dialog.
major	Specifies the component to use to identify the major time axis. Must be a `dvt:timeAxis` component.
menuBar	Specifies the component to use to identify additional controls to appear in the Gantt menu bar. Must be an `af:menu` component
minor	Specifies the component to use to identify the minor time axis. Must be a `dvt:timeAxis` component.
nodeStamp	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 `EditableValueHolder` or `ActionSource` interfaces. Must be an `af:column` component.
tablePopupMenu	Specifies the component to use to identify additional controls to appear in the context menu of the table region. Must be an `af:popup` component.
toolbar	Specifies the component to use to identify additional controls to appear in the Gantt toolbar. Must be an `af:toolbar` component.

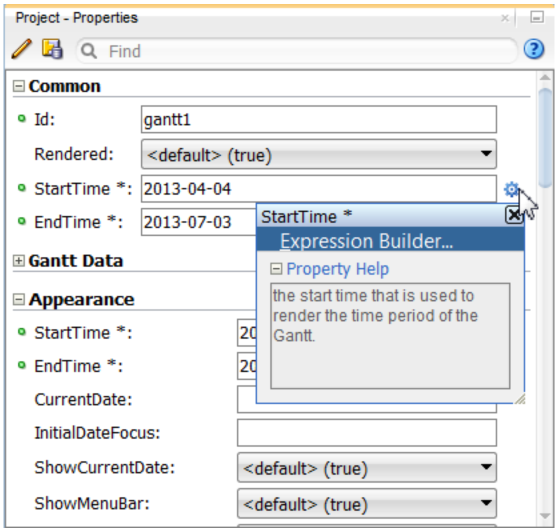
26.2.6 How to Add a Gantt Chart to a Page

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 26–18 shows the dropdown menu for a project Gantt chart component `startTime` attribute.

Figure 26–18 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 Section 26.2.5, "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 Section 26.1.3, "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 Section 22.2.1, "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:

   - **RowSelection**: Use to configure selection for no rows, for a single row, or for multiple rows of the chart region of a Gantt chart.

26.2.7 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. Example 26–2 shows the code inserted in the JSF page for a project Gantt chart.

Example 26–2 Code Sample for 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>
```

26.3 Customizing Gantt Chart Tasks and Resources

Once you have added a 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.

26.3.1 Creating a New Task Type

A task type is represented visually as a bar in the chart region of a Gantt chart. You can create a new task type in one of three ways:

- 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.

Example 26–3 shows sample code to define the properties for a custom task type in the JSF page.

Example 26–3 Sample Code to Define Custom Task Type Properties

```
<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>
```

Example 26–4 shows sample code to define a `TaskbarFormat` object fill and border color and register the object with the `taskbarFormatManager`.

Example 26–4 Custom TaskbarFormat Object Registered with TaskbarFormat Manager

```
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);
```

26.3.2 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 26–3 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. Example 26–5 shows a managed bean that configures the `RUN` metric to stack on the previous metric, `SETUP`.

Example 26–5 Code Sample for Configuring Stacked Bars

```
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;
    }
```

26.3.3 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 26–3 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. Example 26–6 shows the managed bean that configures the resource metric THRESHOLD to step through the vertical bar metrics.

Example 26–6 Code Sample for Configuring Resource Capacity Line

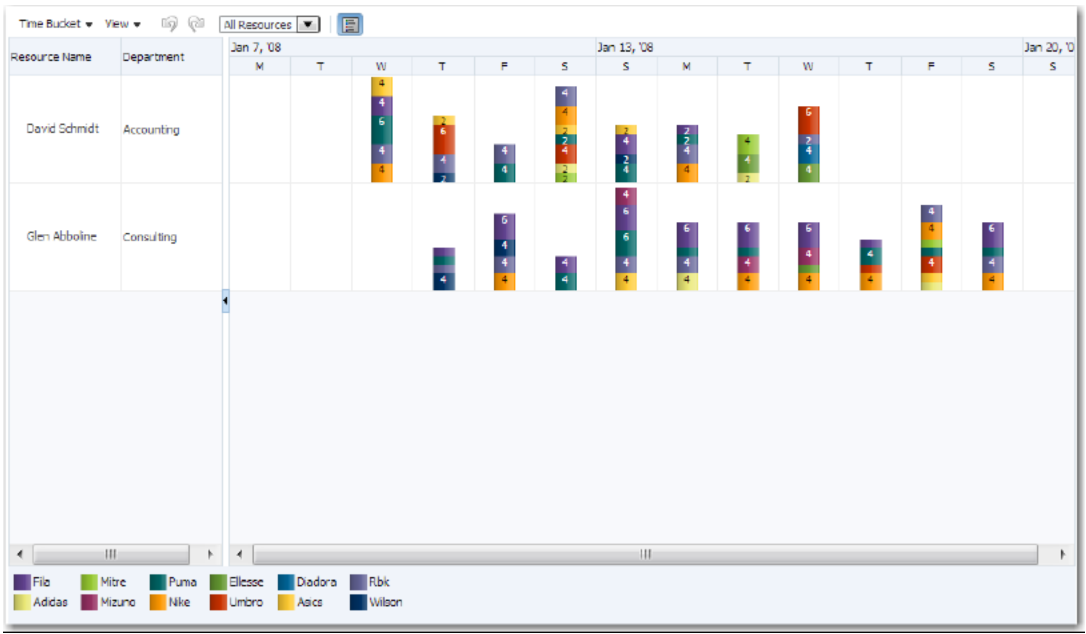
```
public class ResourceUtilizationGanttSteppedLine
{
    private TreeModel m_model;
    public String[] getMetrics()
    {
        return new String[]{"SETUP", "RUN", "AVAILABLE", "THRESHOLD"};
    }
    public TaskbarFormatManager getTaskbarFormatManager()
    {
        TaskbarFormatManager _manager = 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;
    }
```

26.3.4 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 26–19 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 26–19 illustrates the detail values for a `BUDGET` metric by actual values for time spent on each product.

Figure 26–19 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. Example 26–7 shows the sample code for adding the accessor to the page definition file.

Example 26–7 Sample Code for 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>
```

To configure the attribute details for a resource in a resource utilization Gantt chart, use the `setDisplayAs()` method to update the `TaskbarFormat` object. Example 26–8 shows the managed bean that configures the attribute detail metrics for the BUDGET resource bar.

Example 26–8 Code Sample for Configuring Metric Attribute Details

```
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;
    }
}
```

26.4 Customizing Gantt Chart Display Elements

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 Gantt chart legend, customize toolbars and context menus, and configure a custom data filter.

26.4.1 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.

26.4.1.1 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 Section 26.2.5, "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 Section 26.1.3, "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 Section 26.2.6, "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}"`. Example 26–9 shows sample code in a backing bean.

Example 26–9 Backing Bean to Identify Nonworking Days

```
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.

26.4.1.2 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 Section 26.2.5, "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 Section 26.1.3, "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 Section 26.2.6, "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.

26.4.2 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 26–5 shows the valid values and the disabled feature for the Gantt chart types.

Table 26–5 Valid Values for Read-Only Attributes

Value	Feature Disabled
`asListMenu`	Show as List menu item for all Gantt charts.
`asHierMenu`	Show as Hierarchy menu item for all Gantt charts/
`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.
`deleteMenu`	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.

Table 26–5 (Cont.) Valid Values for Read-Only Attributes

Value	Feature Disabled
`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.
`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.

Table 26–5 (Cont.) Valid Values for Read-Only Attributes

Value	Feature Disabled
`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 Gantts.
`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.
`updateTaskMT`	Update Task Edit item, Update Task toolbar item, and right menu items for project and scheduling Gantts.
`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 Section 26.2.5, "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 Section 26.1.3, "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 Section 26.2.6, "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}"`.

26.4.3 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`

Example 26–26 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.

Example 26–10 Gantt Chart Time Axis Set to Months and Weeks

```
<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 Section 26.2.5, "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 Section 26.1.3, "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 Section 26.2.6, "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. Example 26–11 show sample code for the interface.

Example 26–11 Interface to Build Custom Dates

```
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:

   ```
   customTimesScales="#{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. Example 26–12 shows sample code for setting a `threeyears` minor time axis and a `fiveyears` major time axis.

Example 26–12 Custom 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>
```

26.4.4 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.

Example 26–13 show sample code to display information about an On Hold task in the legend of a project Gantt chart.

Example 26–13 Adding a Gantt Chart Legend

```
<dvt:projectGantt var="task">
   <dvt:ganttLegend id="gl" keys="TaskName StartTime EndTime" labels="Name Start
Finish" icons="images/wait.png" iconLabels="OnHold"/>
</dvt:projectGantt>
```

26.4.5 Customizing Gantt Chart Toolbars

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 supply your own menu items and toolbar buttons by using the `menu` and `toolbar` facets in your Gantt chart. The Gantt chart merges the new menu items with the standard items in the Gantt chart. Example 26–14 shows sample code for specifying a new menu item.

Example 26–14 Sample Code for Custom 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>
```

Example 26–15 shows sample code for specifying a new toolbar button.

Example 26–15 Sample Code for Custom 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}"`.

Example 26–16 shows sample code in a backing bean.

Example 26–16 Backing Bean for Handling Data Change

```
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 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*.

26.4.6 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 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. Example 26–17 shows sample code for specifying a custom menu item in the table region context menu.

Example 26–17 Sample Code for Custom Context Menu Item

```
<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. Example 26–18 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.

Example 26–18 Sample Code for Dynamic Context Menu

```
<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. Example 26–19 shows sample code for the backing bean.

Example 26–19 Backing Bean for Handling Task Selection

```
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 Chapter 15, "Using Popup Dialogs, Menus, and Windows".

26.4.7 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`. Example 26–20 shows sample code for registering a Resource filter object with the `FilterManager`.

Example 26–20 Custom Filter Object Registered with 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. Example 26–21 shows sample code for a `FilterListener`.

Example 26–21 FilterListener for Custom Filter

```
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 Section 26.2.5, "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 Section 26.1.3, "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 Section 26.2.6, "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}"`.

26.5 Adding Interactive Features to Gantt Charts

You can add interactive features to Gantt charts 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.

26.5.1 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.

Example 26–22 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.

Example 26–22 Selecting Rows

```
<af:table binding="#{mybean.table}" rowselection="multiple" ...>
  ...
</af:table>
<af:button text="Delete" actionListener="#{mybean.performDelete}"/>
```

Example 26–23 shows an actions method, `performDelete`, which iterates through all the selected rows and calls the `markForDeletion` method on each one.

Example 26–23 Using the rowKey Object

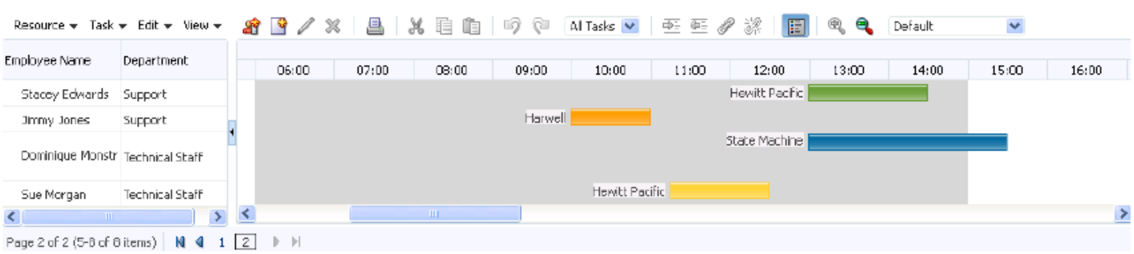
```
public void performDelete(ActionEvent action)
{
  UIXTable table = getTable();
  Iterator selection = table.getSelectedRowKeys().iterator();
  Object oldKey = table.getRowKey();
```

```
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);
}
}
```

26.5.2 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 26–20.

Figure 26–20 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 an alternative page control for the `schedulingGantt`, `projectGantt`, or `resourceUtilizationGantt` component table region, set the `scrollPolicy` attribute to `page`. For example:

```
scrollPolicy="page"
```

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 Appendix D, "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 Section 9.2.1, "Geometry Management and Component Stretching."

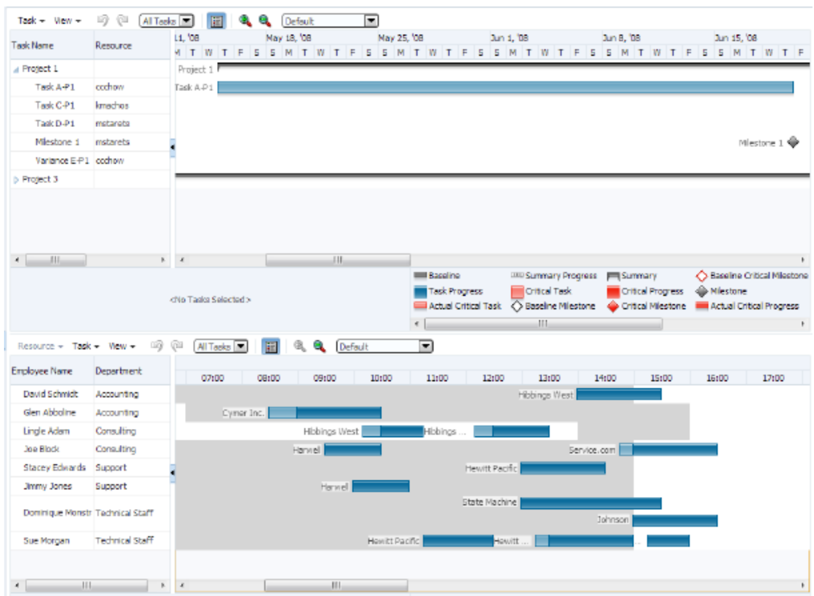
- Set the `scrollPolicy` attribute to `auto` (if the page may also run on a desktop device) or `page` (if the page will only run on a mobile device).

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.

26.5.3 Configuring Synchronized Scrolling Between Gantt Charts

You can configure synchronized horizontal scrolling between the chart side of two Gantt charts. 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 26–21.

Figure 26–21 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. Example 26–24 shows the code for the sychronized scrolling methods. For more information, see Section 4.2, "Adding JavaScript to a Page."

Example 26–24 Code Sample for Sychronized Scrolling Methods

```
<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>
```

Example 26–25 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 Example 26–24.

Example 26–25 Code Sample on JSF Page for Gantt Charts with Synchronized Scrolling

```
<dvt:projectGantt id="gantt1" 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 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>
```

26.5.4 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.

26.5.4.1 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.

  Example 26–26 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.

Example 26–26 Code for Setting the Time Period Option for Printing a Gantt Chart

```
_printer.setStartTime(startDate);
_printer.setEndTime(endDate);
```

26.5.4.2 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.

Example 26–27 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.

Example 26–27 Sample ActionListener for Handling the Gantt Chart Print Event

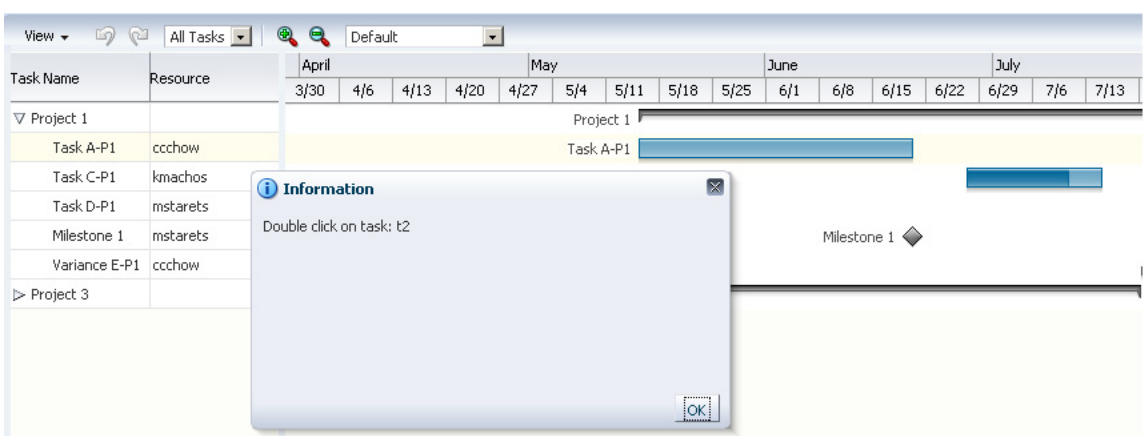
```
public void handleAction(GanttActionEvent evt)
{
```

```
    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");
     }
  }
```

26.5.5 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 26–22 shows a project Gantt chart with a double-click event on a task bar.

Figure 26–22 Task Bar with Double-Click Event



Example 26–28 show sample code for adding a double-click event to a task bar.

Example 26–28 Sample Code for Double-Click Event

```
<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:

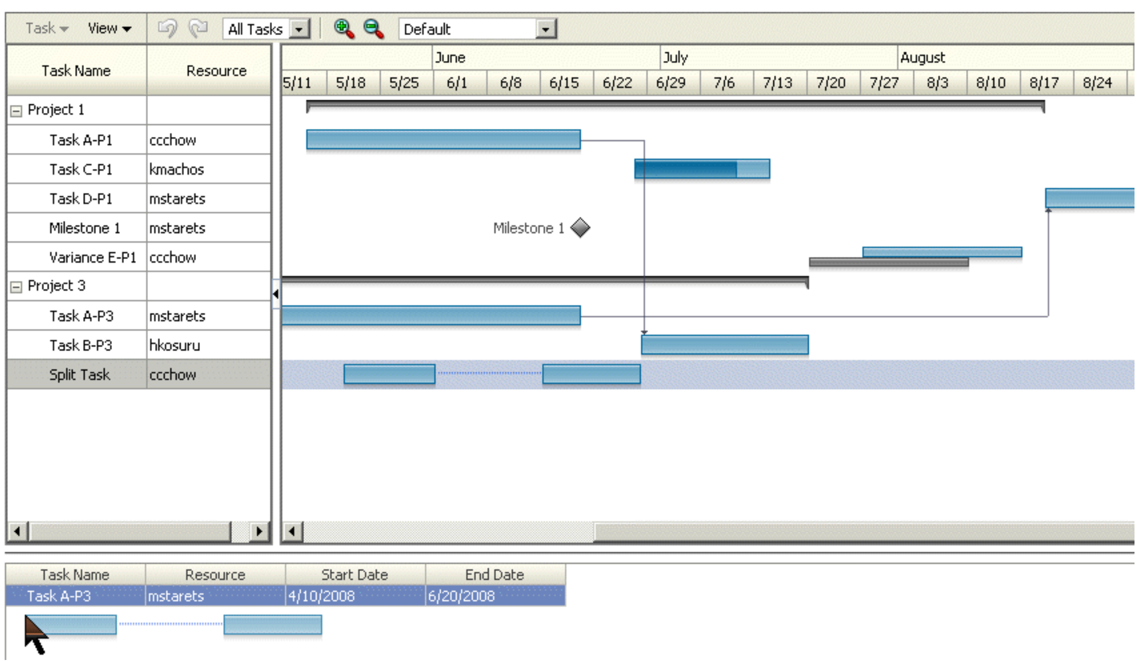
```
public void handleDoubleClick(DoubleClick event)
```

26.5.6 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 26–23 shows a project Gantt chart with tasks dragged from the table region onto a table of tasks.

Figure 26–23 Project Gantt Chart as Drag Source



Example 26–29 shows sample code for adding drag and drop functionality to a project Gantt chart.

Example 26–29 Sample Code for Adding Drag and Drop Functionality

```
<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>
      </dvt:projectGantt>
```

Example 26–30 shows sample code for the listener method for handling the drop event.

Example 26–30 Event Handler Code for a dropListener for a Collection

```
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 Section 36.4, "Adding Drag and Drop Functionality for Collections".

27

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:

27.1 About Timeline Components

A *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 a links. A dual timeline can be configured to display two series of events to allow a side-by-side comparison of related information.

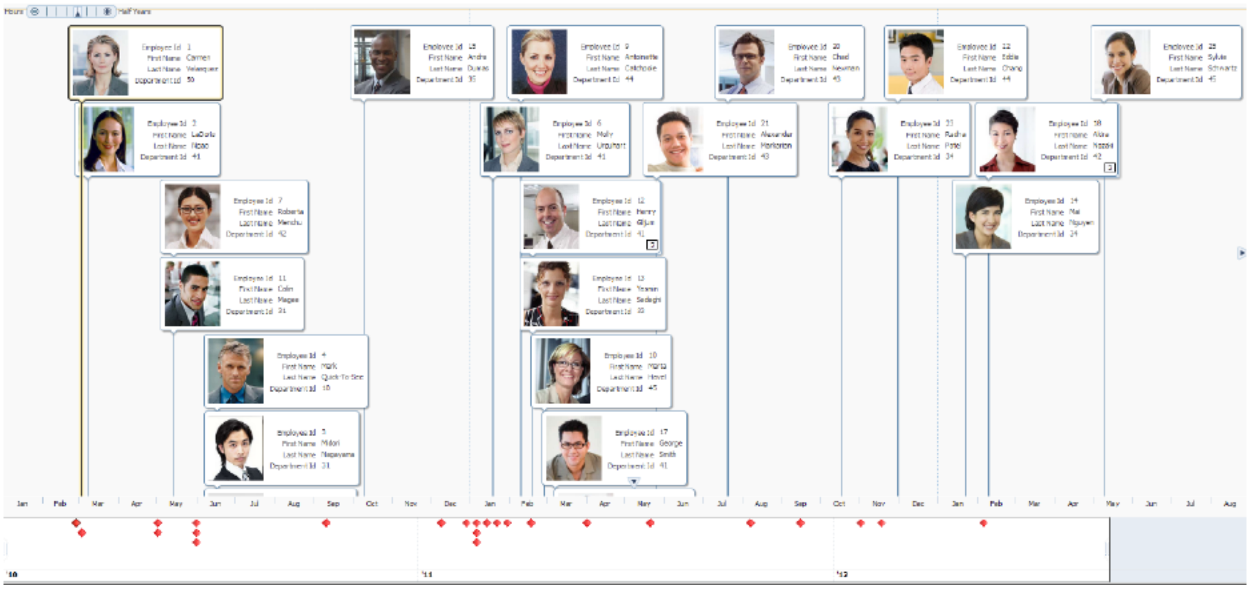
The timeline component supports expanding and collapsing a group of related timeline items, such as shared hire dates, or a group of related activities such as completion of a number of employee forms. The timeline component also supports an adjustable time range to change the view for zooming in or out.

27.1.1 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.

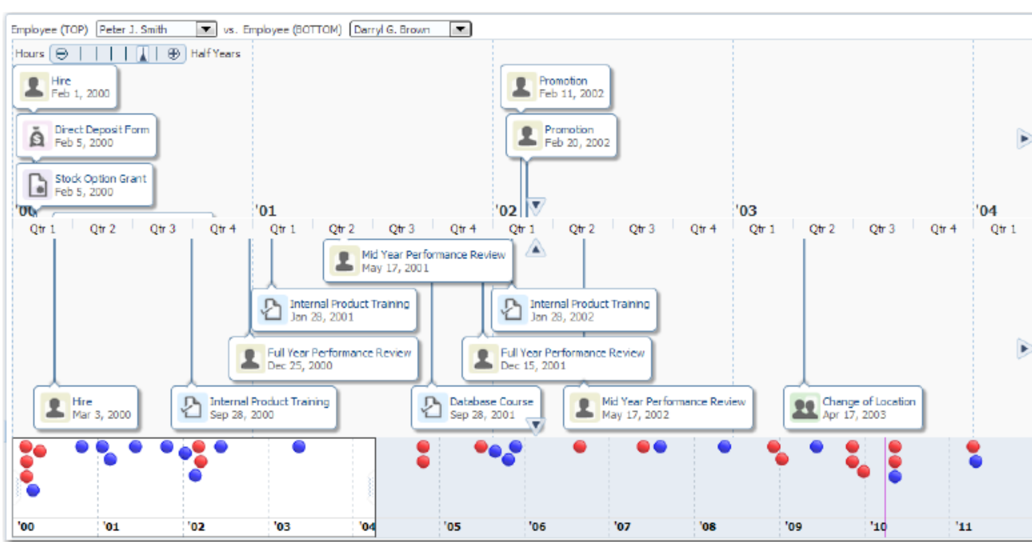
For example, the timeline in Figure 27–1 is configured to display the chronological order of the hire dates of employees in the Summit DVT example. In this example, timeline items representing each event display information about the employee using an image and text with labels. The overview window defines 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 27–1 Timeline of Employee Hire Dates



A dual timeline can be used for comparison of up to two series of events. Figure 27–2 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 27–2 Dual Timeline Comparing Employee Change Events



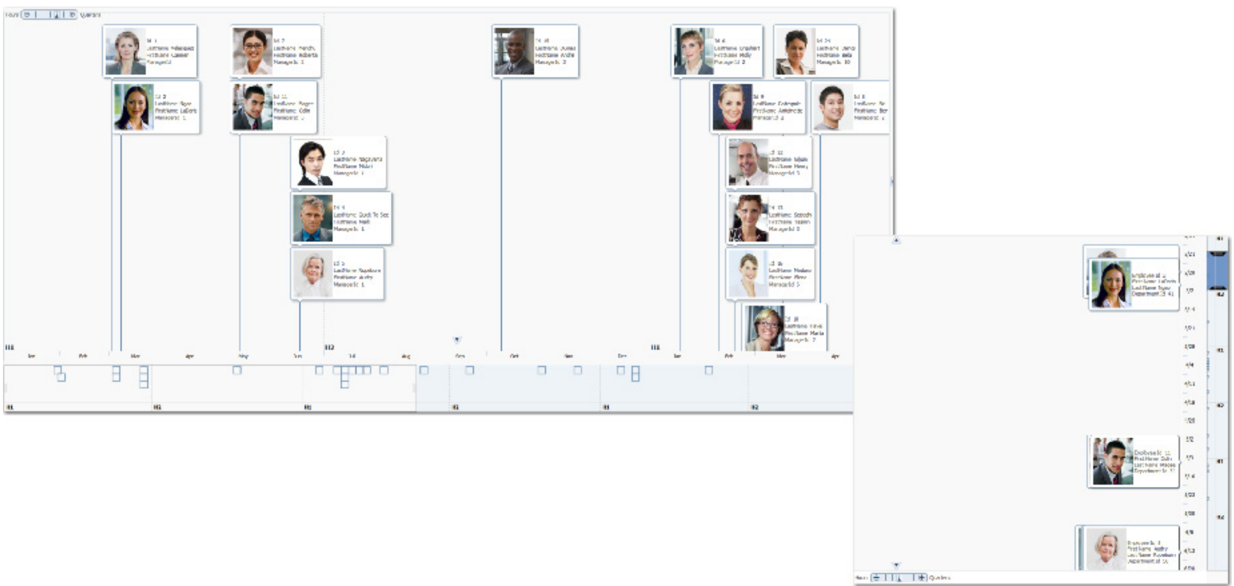
27.1.2 End User and Presentation Features

To understand how timelines are used and can be customized, it is helpful to understand these elements and features.

27.1.2.1 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 27–3 illustrates the comparison of a timeline using a horizontal orientation with the same timeline using a vertical orientation.

Figure 27–3 Timeline Horizontal and Vertical Orientations



27.1.2.2 Timeline Item Selection

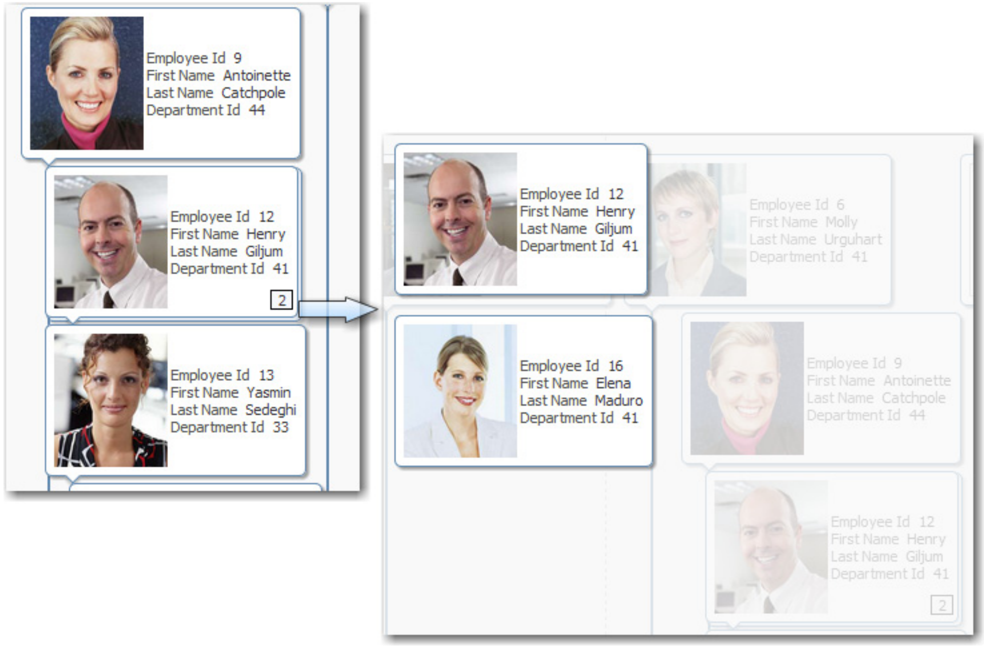
Each event displayed in the timeline is represented as a timeline item that can include data display components such as images, text, and text labels, or actions such as links, buttons, and menus. A line feeler connects the event to the date in the time axis of the timeline. Events are represented in 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.

27.1.2.3 Timeline Grouping and Sorting

Timeline items that share a common date can be configured to display as a group that can be expanded or collapsed at runtime. By default, a number counter displaying the number of items in a group is provided in the collapsed view. Clicking anywhere in the grouped timeline item opens all items in the collapsed view and clicking in the timeline collapses the expanded view. Figure 27–4 shows a timeline item with a counter opened into an expanded view.

Figure 27–4 Timeline Group Counter and Expanded View

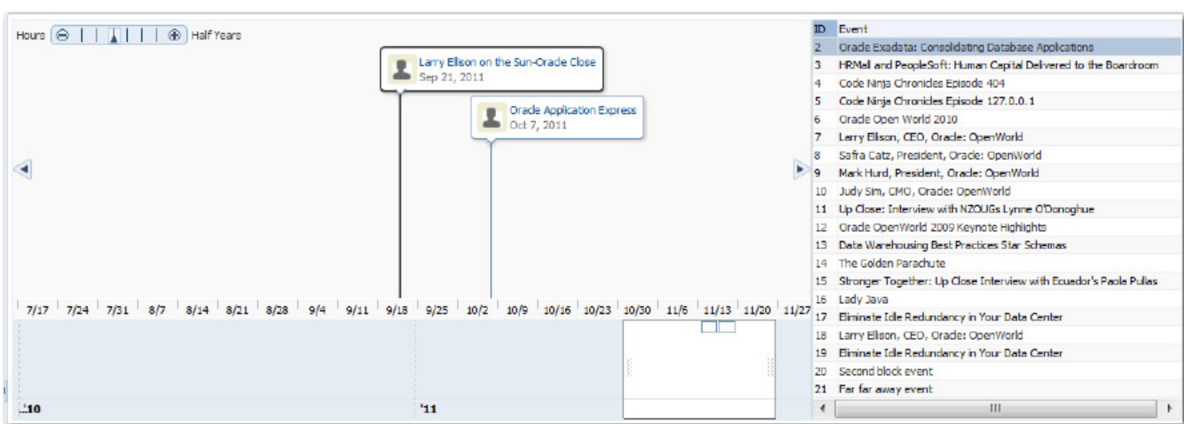


In the default horizontal orientation of the timeline, an overview panel is displayed at the bottom of the timeline. The overview panel includes a movable overview window that corresponds to the period of viewable time in the timeline, and an overview time axis that displays the total time increment for the timeline.

27.1.2.4 Drag and Drop Support

Timeline components support drag and drop operations to and from another collection component, for example, a table. Figure 27–5 shows a timeline configured as a drop target and drag source. When the user drags one of the rows in the table onto the timeline, attributes are displayed as a timeline item. Timeline items can also be selected and dragged to the table to display attributes on a row.

Figure 27–5 Timeline Configured as a Drop Target and Drag Source



27.1.2.5 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.

27.1.2.6 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 Section A.4.3, "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

27.1.2.7 Timeline 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 Chapter 37, "Using Different Output Modes."

When a timeline is displayed on a JSF page to be output in printable or emailable 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.

27.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: By default, timeline components are accessible. You can configure your application pages with timeline components to be accessible to screen reader users. For more information, see Chapter 33, "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 Appendix D, "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 Section 12.2.2, "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 Section 22.2, "Common Functionality in Data Visualization Components."

27.2 Using Timeline Components

To use the `timeline` component in UI-first development, define the data, add the timeline to a page and complete the additional configuration in JDeveloper.

27.2.1 Timeline Component Data Requirements

The data layer for the `timeline` component is specified in its child, the `timelineSeries` component. You must specify at least one timeline 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 more 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 `timelineSeries` 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 `timelineSeries` component must be at most one `timelineItem` component. This component makes it possible to customize the event content through stamping. When you use stamping, child components are not created for every event represented in a timeline. Rather, the content of the component is repeatedly rendered, or stamped, once per timeline item, such as the events in the timeline.

Each time a timeline item is stamped, 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 timeline item component, for example, to pass the item value to a stamped `af:outputText` component. 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.

Example 27–1 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.

Example 27–1 Managed Bean Example to Specify Timeline Data

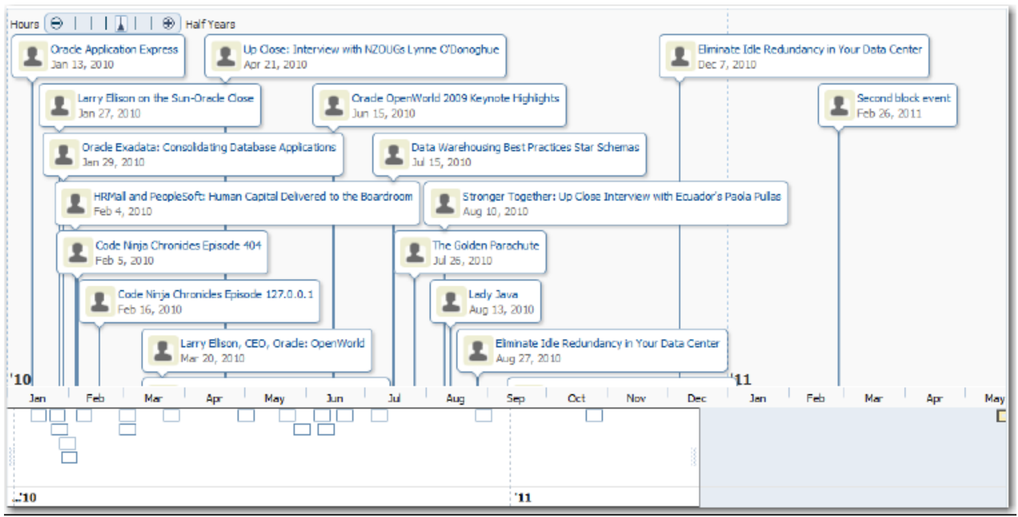
```
  //imports needed by methods
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.Set;
```

```
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.UITimelineSeries;
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", "se198AyXcsk", 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();
    UITimelineSeries _series =
      (UITimelineSeries)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 27–6. You can find the complete source code for the TimelineCBBean in the ADF Faces demo application. For more information about the demo application, see Chapter 2, "ADF Faces Demo Application."

Figure 27–6 Timeline of Employee Presentations



27.2.2 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 (`timelineSeries`): 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 (`timelineItem`): The child of `timelineSeries` that represents an event in the timeline. The components supports the use of many ADF Faces components, such as `af:outputText`, `af:image`, and `af:panelGroupLayout`.

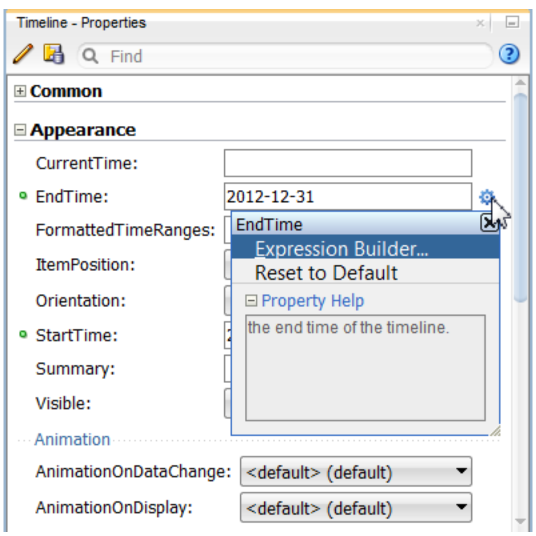
- Marker (`marker`): A configurable shape that represents the event in the overview panel. The attributes are specified in a named `overviewItem` facet child of the `timelineItem` component.
- Time axis (`timeAxis`): Child of `timeline` used to specify the time axis and `timelineOverview` used to specify 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 sccroll through the timeline using a zoom control.

27.2.3 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 27–7 shows the dropdown menu for a timeline `endTime` attribute.

Figure 27–7 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 "How to Configure Databound Graphs for Drilling" 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 Section 27.2.2, "Configuring Timelines."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 27.1.3, "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 Section 27.1.2.1, "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 authomatically 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 a timeline overview 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`.

27.2.4 What Happens When You Add a Timeline to a Page

JDeveloper generates only a single 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. Example 27–2 shows the generated code.

Example 27–2 Timeline Sample Code in UI-First Development

```
<dvt:timeline
  startTime="2012-06-27" endTime="2012-06-27" id="t1"/>
```

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. For more information, see the "Creating Databound Gantt Chart and Timeline Components" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

27.3 Adding Data to Timeline Components

Timeline components require a collection data model to display attributes. For example, to create the Employee Presentation timeline illustrated in Figure 27–6, you must provide a data model that includes a qualifying date value and details about the events.

27.3.1 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. For more information, see Section 27.2.2, "Configuring Timelines."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 27.1.3, "Additional Functionality for Timeline Components."

Add a timeline to your page. For help with adding a timeline to a page, see Section 27.2.3, "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 additional information and examples, see Section 27.2.1, "Timeline Component Data Requirements." 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 Section 3.6, "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** > **Timeline Series**.
2. Right-click the `timelineSeries` 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}`.

     For help with creating EL expressions, see Section 3.5.1, "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 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 `timelineSeries` node and choose **Insert Inside Timeline Series** > **Timeline Item** to add a component to display the timeline series data through stamping.
5. In the Properties window for the `dvt:timelineItem`, 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}`.

- **Group**: Optionally, you can configure timeline items that share a common date to display as a group that can be expanded or collapsed at runtime. By default, a number counter displaying the number of items in a group is provided in the collapsed view.

6. To configure the timeline item to display the data collection attributes in the timeline item, do the following:

   a. Use the Structure window context menu to insert components to define the layout of the timeline item.

   b. Use the Properties window to specify the content and display attributes for the timeline item. For the `value` attribute use an EL Expression that references the row in the data collection.

      For example, the code highlighted in Example 27–3 shows the component structure and attribute definitions for the timeline item stamped in the employee presentations timeline in Figure 27–6.

7. To configure the marker representing the timeline item that displays in the timeline overview, 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.

> **Note:** You can specify the layout and contents of a timeline item in a number of ways. For more information, see Section 27.4.1, "Customizing Timeline Items."

Example 27–3 Code Sample for Timeline Bound to Data

```
<dvt:timeline id="tl1" startTime="2010-01-01" endTime="2011-12-31"
    inlineStyle="width:1000px;height:500px" itemSelection="single">
  <dvt:timelineSeries id="ts1" var="evt" value="#{cb.model}">
    <dvt:timelineItem id="ti1" value="#{evt.date}">
      <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>
  <dvt:timeAxis id="ta1" scale="weeks"/>
  <dvt:timelineOverview id="ov1">
```

```
    <dvt:timeAxis id="ta2" scale="years"/>
  </dvt:timelineOverview>
  <f:ajax event="keyUp" execute="@this" listener="#{cb.handleKey}"/>
</dvt:timeline>
```

27.3.2 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 `timelineSeries` component is the same.

27.3.3 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*.

27.4 Customizing Timeline Display Elements

You can configure timeline items and add a custom time scale to your timeline.

27.4.1 Customizing Timeline Items

Timeline items represent the events displayed in the timeline. The `timelineItem` component supports the following ADF components to display information and provide actions associated with the event:

- Layout components including: `af:panelFormLayout`, `af:panelGroupLayout`, `af:separator`, `af:showDetailItem`, and `af:spacer`. For more information about using these components, see Chapter 9, "Organizing Content on Web Pages."
- Menu component `af:menu`. For more information about these components, see Chapter 16, "Using Menus, Toolbars, and Toolboxes."
- Output components including: `af:outputFormatted` and `af:outputText`. For more information about these components, see Chapter 18, "Using Output Components."
- Message component `af:outputLabelMessage`. For more information about this component, see Chapter 19, "Displaying Tips, Messages, and Help."
- Navigation components including: `af:button` and `af:link`. For more information about these components, see Chapter 20, "Working with Navigation Components."
- Image component `af:image`. For information about how to use the `af:image` component, see Section 18.4, "Displaying Images."

- `af:showPopupBehavior`: For information about how to use the `af:showPopupBehavior` component, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

Timeline items are represented in the timeline overview as a configurable shape. You can specify the following attributes for a timeline 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.

27.4.2 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 Section 27.2.2, "Configuring Timelines."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 27.1.3, "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 Section 27.2.3, "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. Example 27–4 show sample code for the interface.

Example 27–4 Interface to Build Custom Dates

```
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:

   ```
   customTimesScales="#{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.

   Example 27–5 shows sample code for setting a `threeyears` time axis and a `fiveyears` overview time axis.

Example 27–5 Custom Time Axis

```
<dvt:timeline>
  <dvt:timeAxis id="ta1" scale="threeyears"/>
  <dvt:timelineOverview id="ov1">
     <dvt:timeAxis id="ta2" scale="fiveyears"/>
  </dvt:timelineOverview>
</dvt:timeline>
```

27.5 Adding Interactive Features to Timelines

You can add interactive features to timelines, including support for popups, custom context menus, and drag and drop operations.

27.5.1 How to Add Popups to Timeline Items

Timeline `timelineItem` 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 Chapter 15, "Using Popup Dialogs, Menus, and Windows."

27.5.2 How to Configure Timeline Context Menus

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 on a JSP or JSPX page supports a single child component. For these facets to work, selection must be enabled in the timeline’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 Chapter 15, "Using Popup Dialogs, Menus, and Windows."

27.5.3 How to Add Drag and Drop to a Timeline

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 (for example, a row from a table), and drop it into a timeline, or drag an event from a timeline and drop it into a table, as illustrated in Figure 27–5.

To add drop support to a timeline, add the `af:dropTarget` tag to the `timeline` component and include the data flavors that the timeline will support. Add a `dropListener` method to a timeline managed bean that will respond to the drop event.

To add drag support from a timeline to a collection component, add the `af:dragSource` tag to the `timeline` component and add the `af:collectionDropTarget` 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:collectionDropTarget` and also define a `dropListener` method to respond to the drop event.

Example 27–6 shows the JSF page sample code for the ADF Faces demo application illustrated in Figure 27–5. For additional information about the `af:table` component, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."

Example 27–6 Sample Code for Timeline Drop Target and Drag Source

```
<dvt:timeline id="tl1" 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>
  <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:timelineSeries>
  <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>
```

The data model for this example is defined in the `TimelineDnDBean` managed bean using an `ArrayList` class. You can find the source code for the class and the supporting `EmpEvent` class in the ADF Faces demo application. For more information about the demo application, see Chapter 2, "ADF Faces Demo Application."

Before you begin:

It may be helpful to have an understanding of how timeline attributes and child tags can affect functionality. For more information, see Section 27.2.2, "Configuring Timelines."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 27.1.3, "Additional Functionality for Timeline Components."

You will need to complete these tasks:

- Add a timeline to your page. For more information, see Section 27.2.3, "How to Add a Timeline to a Page"
- If you are configuring timeline items as a drag source and you did not bind the timeline to a data control when you added the component to the page, add data to the timeline. For information about adding data to timelines using UI-first development, see Section 27.3.1, "How to Add Data to a Timeline."
- Create any additional components needed to support the drag and drop.

  For example, if you are using a table as the drag source or drop target, you will need to add a table to your 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.

   Example 27–7 shows the sample drop listener and supporting methods for the timeline displayed in Figure 27–5.

Example 27–7 Sample Drop Listener for a Timeline

```
// 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 of the `discriminant` attribute of the `af:dragSource` component you will set for the collection component receiving the drags from the timeline in Step 5.

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 of the `modelName` attribute of the `af:collectionDropTarget` component you will set for the collection component receiving the drags from the timeline in Step 8.

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.

      Example 27–8 shows the sample drop listener for the timeline displayed in Figure 27–5. This example uses the same imports and helper methods used in Example 27–7, and they are not included here.

Example 27–8 Sample Drop Listener for a Table Using a Timeline as a Drag Source

```
//Drop Listener
public DnDAction handleDropOnTable(DropEvent event)
{
  Integer _dropSite = (Integer)event.getDropSite();
  Transferable _transferable = event.getTransferable();
  RowKeySet _rowKeySet = _transferable.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 you set in Step 7.

For more information about configuring drag and drop on ADF Faces or ADF Data Visualization components, see Chapter 36, "Adding Drag and Drop Functionality."

28

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:

28.1 About Map Components

A geographic map represents business data in one or more interactive layers of information (known as *themes*), superimposed on a single map. 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.

A *thematic map* represents business data as patterns in stylized areas or associated markers and does not require a connection to an Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map.

28.1.1 Map Component Use Cases and Examples

Both geographic and thematic maps are designed to display data. The difference is that 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 28–1 displays a geographic map with color, point, and graph themes.

Figure 28–1 Geographic Map of Southwest US with Color, Point, and Pie Graph Themes

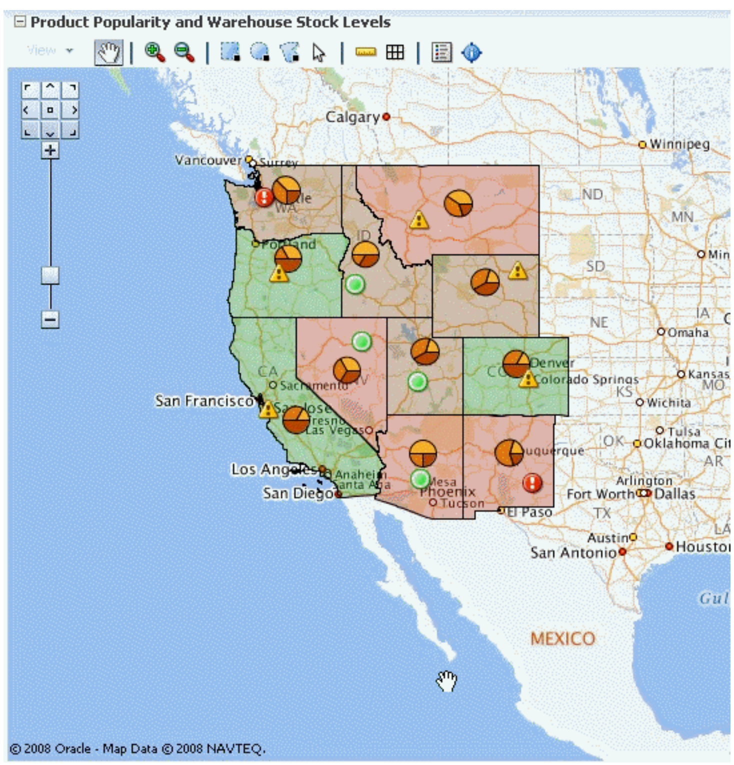
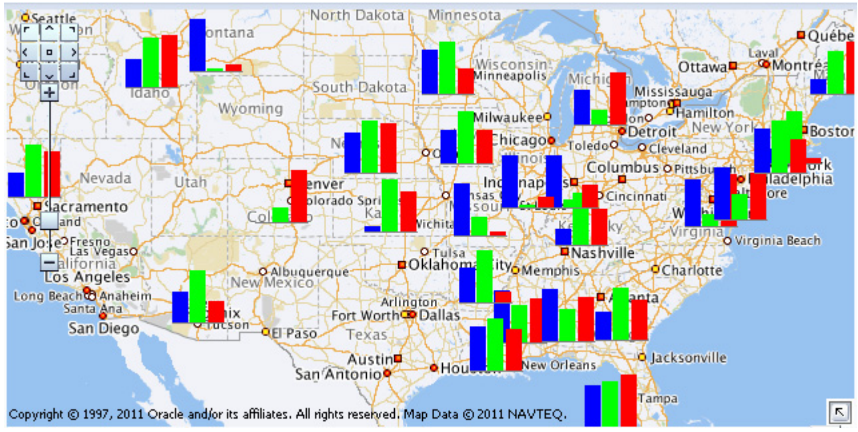


Figure 28–2 shows a geographic map with a customized bar graph theme.

Figure 28–2 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 28–3 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 28–3 Thematic Map of Unemployment Rates in the US

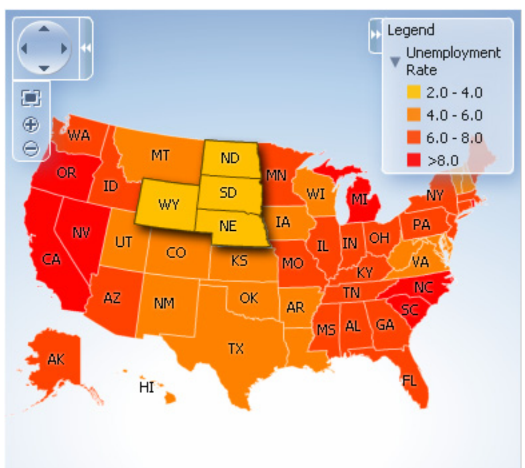


Figure 28–4 shows a thematic map that displays a graduated symbol for the size of the major cities in South America.

Figure 28–4 Thematic Map with City by Size in South American

28.1.2 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.

28.1.2.1 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 28–5 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.

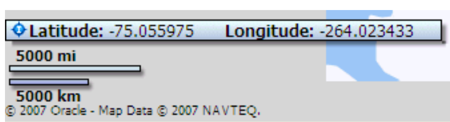
  You can customize the location and the initial setting of the zoom control in the `map` component. The **View** menu in the map toolbar lets you determine the visibility of the zoom control. By default, the initial zoom level for a map is set to 0.

Figure 28–5 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.

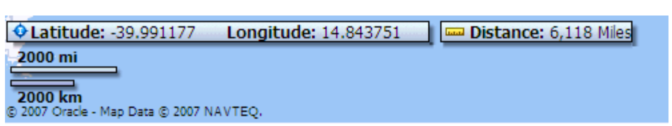
- Scale: Two horizontal bars that display in the lower left-hand corner of the map below the information panel and above the copyright. Figure 28–6 shows the scale. The top bar represents miles (mi) and the bottom bar represents kilometers (km). Labels appear above the miles bar and below the kilometers 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 28–6 Map Information Panel, Scale, and Copyright



- Information panel: Displays latitude and longitude in the lower left-hand corner above the scale. Figure 28–6 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 28–7 shows the measurement panel with a distance measurement. Area measurement and radius measurement appear in a similar manner with the appropriate labels.

Figure 28–7 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 28–8. 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 28–8 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.

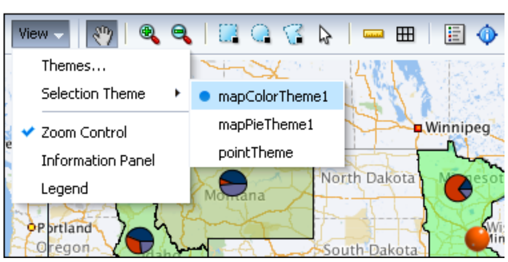
– Show/Hide icon: Appears in the upper left-hand corner when the overview map is displayed. When you click the **Show/Hide icon**, the overview map becomes invisible and only the icon can be seen in the lower right corner of the main map.

- 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:

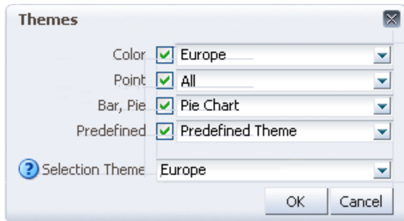
– View menu: Lets the user control which themes are visible, select a specific theme for display, and determine the visibility of the zoom control, information panel, and the legend. Figure 28–9 show a sample **View** menu.

Figure 28–9 Map View Menu



The **Themes** option on the **View** menu provides a dialog to configure a geographic map when multiple themes are available. Figure 28–10 shows a sample map themes dialog.

Figure 28–10 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

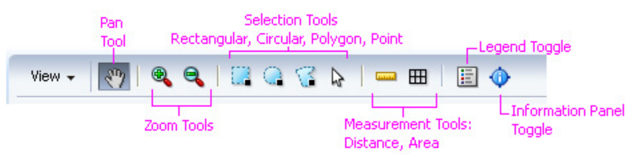
  * Measurement: Use to hide or show the map distance and area measurements tools

  * Legend: Use to hide or show the map legend

  * Information panel: Use to hide or show the map information panel

  Figure 28–11 shows the features supported by the map toolbar buttons

Figure 28–11 Map Toolbar Button Features



28.1.2.2 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

- 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:

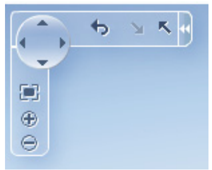
  - Area data layers: Area definitions associated with a geographic area, or by points associated with a map position as in longitude and latitude, or x an y coordinates. Data is stylized with color and pattern fills, or a data marker, or both.
  - Point data layers: Point locations associated with a map 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 28–12 shows the control panel with drilling enabled in the map.

Figure 28–12 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 28–13 shows the default context menu for the map viewport. The same menu items are available for data markers.

Figure 28–13 Map Viewport Background Context Menu

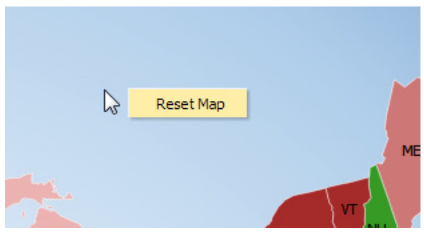
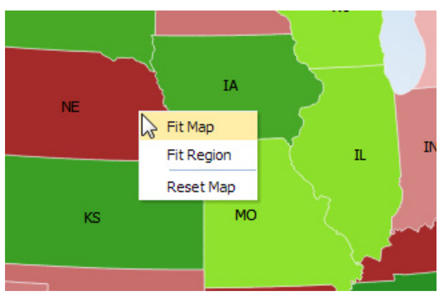


  Figure 28–14 shows the context menu for a map region.

Figure 28–14 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.

28.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: By default, geographic and thematic map components are accessible. You can make your application pages accessible to screen readers. For more information, see Chapter 33, "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 Chapter 31, "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 Section 12.2.2, "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 Section 22.2, "Common Functionality in Data Visualization Components."

28.2 Using the Geographic Map Component

When you create a 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.

28.2.1 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 adminstrator 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.

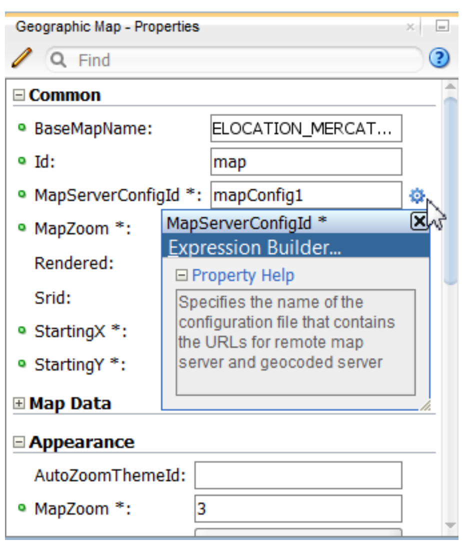
28.2.2 How to Add a Geographic Map to a Page

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. shows the dropdown menu for a thematic map component `mapServerConfigId` attribute.

Figure 28–15 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 Section 28.2.1, "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 Section 28.1.3, "Additional Functionality for Map Components."

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:

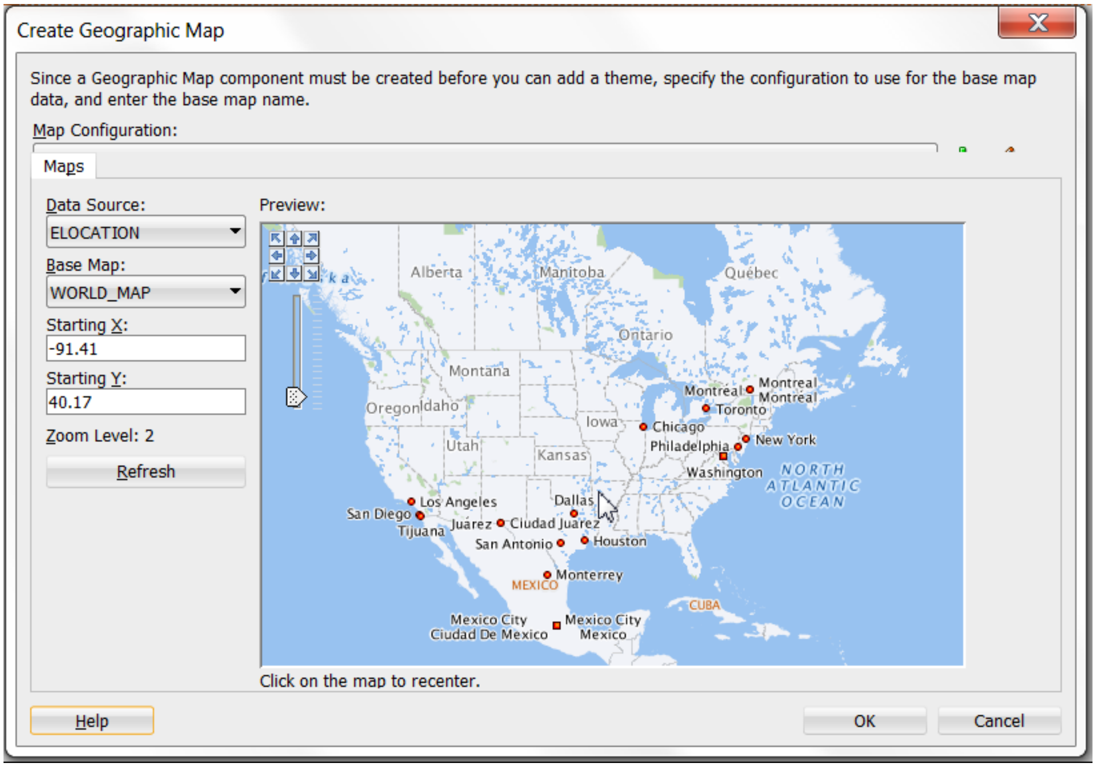
   - Use the dropdown list to choose an available configuration, or
   - Click the Add icon to open the Create Geographic Map Configuration dialog.

     In the dialog you can specify the **MapViewer 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 28–16 shows a sample completed dialog.

Figure 28–16 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.

   - **AutoZoomThemeId**: 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 Section 28.3.3, "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 **AutoZoomThemeId** are visible. If the **AutoZoomStrategy** is set to **CENTERATZOOMLEVEL**, the map will center on the theme in AutoZoomThemeId, 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.

28.2.3 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. Example 28–1 shows the code added to the JSF page.

Example 28–1 Geographic Map Added to 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 Chapter 15, "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*.

28.2.4 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 Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

28.3 Customizing Geographic Map Display Attributes

You can customize geographic display attributes including 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.

28.3.1 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 Section 28.2.1, "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 Section 28.2.2, "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`.

28.3.2 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 Section 28.2.1, "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 Section 28.2.2, "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:

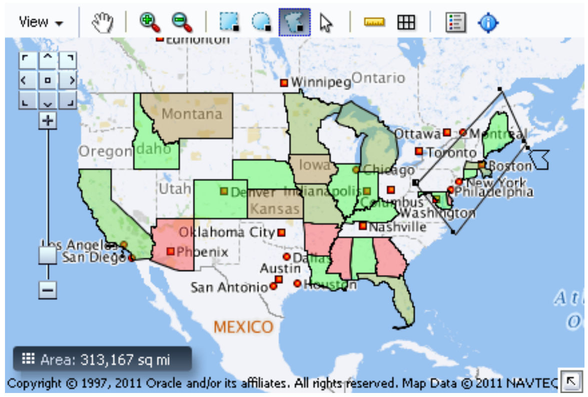
1. In the Structure window, right-click the **dvt:map** component and choose **Go to Properties**.
2. In the Properties window, expand the **Appearance** section. Use this section to set the following attributes:
   - **AutoZoomThemeID**: Enter the Id of the first theme that will be displayed.
   - **ZoomBarStrategy**: 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`.

28.3.3 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 28–17 show the use of the polygon tool to select an area in a map with a color theme.

Figure 28–17 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 Section 28.2.1, "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 Section 28.2.2, "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.

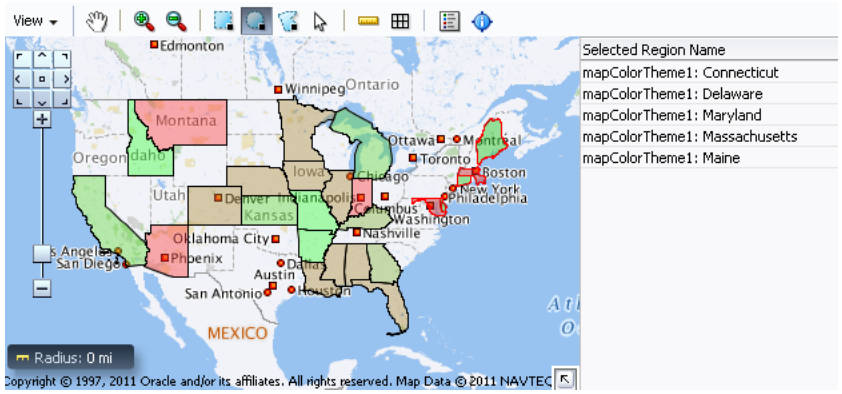
To customize map selection tool attributes:

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 28–18 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 28–18 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. Example 28–2 shows sample code for a backing bean.

Example 28–2 Sample Code in Backing Bean for Selection Listener

```
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**.

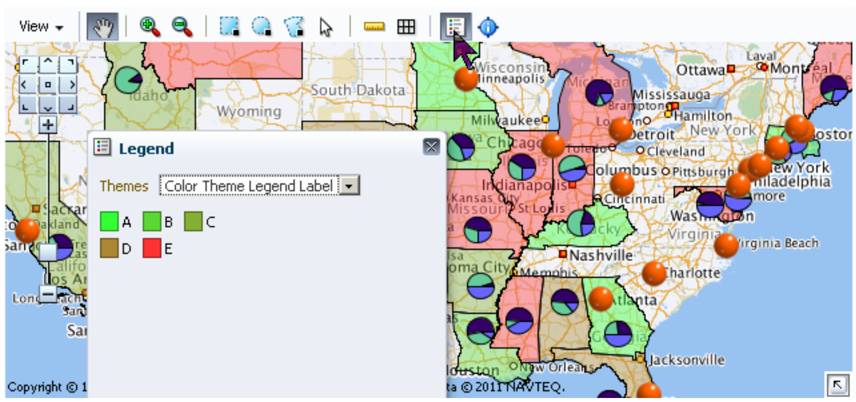
2. In the Properties window, expand the **Behavior** section. For the **SelectionListener** attribute, enter a method reference that points to the backing bean. For example:

```
#{eventBean.processSelection}
```

28.3.4 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 28–19 shows a map displaying the legend for one of its multiple themes.

Figure 28–19 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 Section 28.5, "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 Section 28.2.1, "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 Section 28.2.2, "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.

28.4 Customizing Geographic Map Themes

Each of the geographic map themes, color, point, pie graph, and bar graph, 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.

28.4.1 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 Section 28.2.1, "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 Section 28.2.2, "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.

28.4.2 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 Section 28.2.1, "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 Section 28.2.2, "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"
   ```

28.4.3 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 Section 28.2.1, "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 Section 28.2.2, "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.

28.4.4 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 Section 28.2.1, "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 Section 28.2.2, "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.

28.4.5 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. Example 28–3 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.

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.

Example 28–3 Map Point Theme Code with Custom Point Images

```
<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>
```

28.4.6 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 Section 28.2.1, "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 Section 28.2.2, "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 hexidecimal 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.

28.4.7 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. Example 28–4 shows sample source code generated on the JSF page when you customize the bars in a map bar graph.

Example 28–4 Code for Customizing 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>
```

28.4.8 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 Section 28.2.1, "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 Section 28.2.2, "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 hexidecimal 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.

28.4.9 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. Example 28–5 shows sample code generated in a JSF page when you customize the slices in a map pie graph.

Example 28–5 Code for Customizing 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>
```

28.5 Adding a Toolbar to a Geographic Map

When you create a 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 28–20 shows a map toolbar.

Figure 28–20 Geographic Map Toolbar

For more information about toolbar functionality, see Section 28.1.2.1, "Geographic Map End User and Presentation Features."

28.5.1 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 Section 28.2.1, "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 Section 28.2.2, "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**.

28.5.2 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.

Example 28–6 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.

Example 28–6 Code Generated for a Map Toolbar

```
<af:form>
  <dvt:mapToolbar mapId="map_us" id="T1"/>
  <dvt:map  id="map_us"
   ...
  </dvt:map>
</af:form>
```

28.6 Using Thematic Map Components

To display data in 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.

28.6.1 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 Section 28.7.3, "How to Format Numeric Data Values in Area and Marker Labels."

Figure 28–21 illustrates the basic tag structure for configuring a data layer in a thematic map.

Figure 28–21 Tag Structure for Thematic Map Area Data Layers

```
<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:area> or </dvt:marker> or </af:image>
         </dvt:areaLocation>
     </dvt:areaDataLayer>
   </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 Section 28.7.3, "How to Format Numeric Data Values in Area and Marker Labels."

Figure 28–22 illustrates the basic tag structure for configuring a point data layer in a thematic map.

Figure 28–22 Tag Structure for Configuring Thematic Map Point Data Layer

```
<dvt:thematicMap basemap="usa">
  <dvt:areaLayer layer="states">
    <dvt:pointDataLayer id="pdl1" 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: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.

Figure 28–23 illustrates the tag structure for nesting point data layers. In the illustration, point data layer `pd1` is only displayed when the `states` layer is displayed. Point data layer `pd2` is always displayed.

Figure 28–23 Tag Structure for Nesting Thematic Map Point Data Layers

```
<dvt:thematicMap basemap=“usa”…>
  <dvt:areaLayer  layer=“states”>
    <dvt:areaDataLayer ..../>
    <dvt:pointDataLayer id=“pdl1”…/>
 </dvt:areaLayer>
 <dvt:areaLayer  layer=“counties”/>
 <dvt:pointDataLayer id=“pdl2”…>
</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.

28.6.2 What You May Need to Know About 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 28–1 shows the valid map layers for each base map.

Table 28–1 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*.

28.6.3 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.

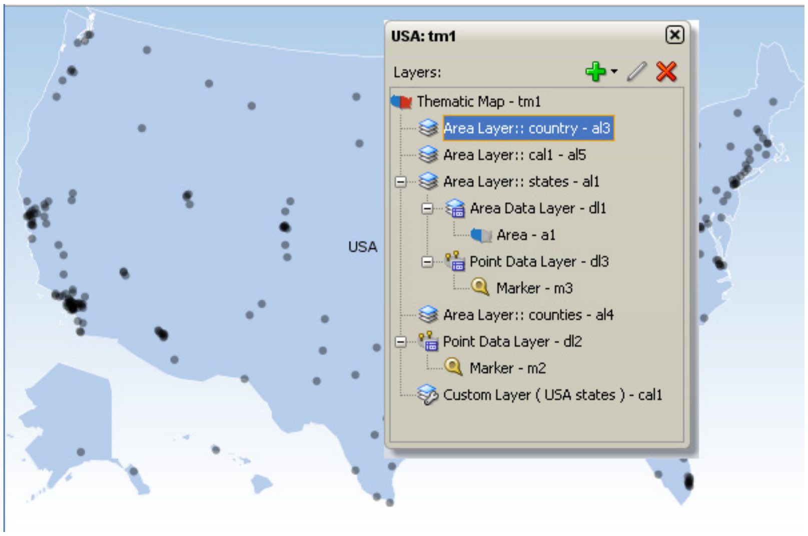
The Layer Browser toolbar provides controls for the following operations:

- 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 `al1`, `al2`, `al3`, and so on. Custom layers (`customAreaLayer`) are assigned `cal1`, `cal2`, `cal3`, and then referenced by an `areaLayer` component within the consecutive order. Data layers including area (`areaDataLayer`) and point (`pointDataLayer`) components are assigned `dl1`,`dl2`, `dl3`, 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 28–23 shows a Layer Browser displaying the hierarchical structure of a thematic map.

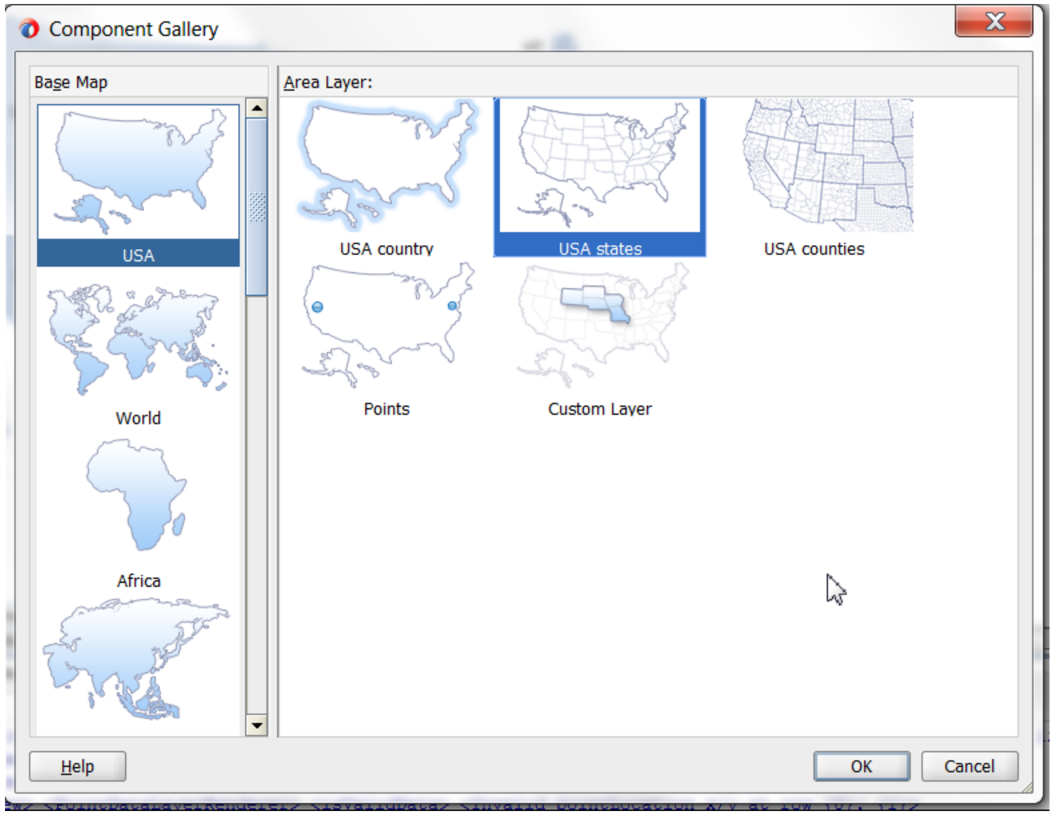
Figure 28–24 Thematic Map Layer Browser



28.6.4 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 28–25 show the Component Gallery for thematic maps with the United States base map and states layer selected.

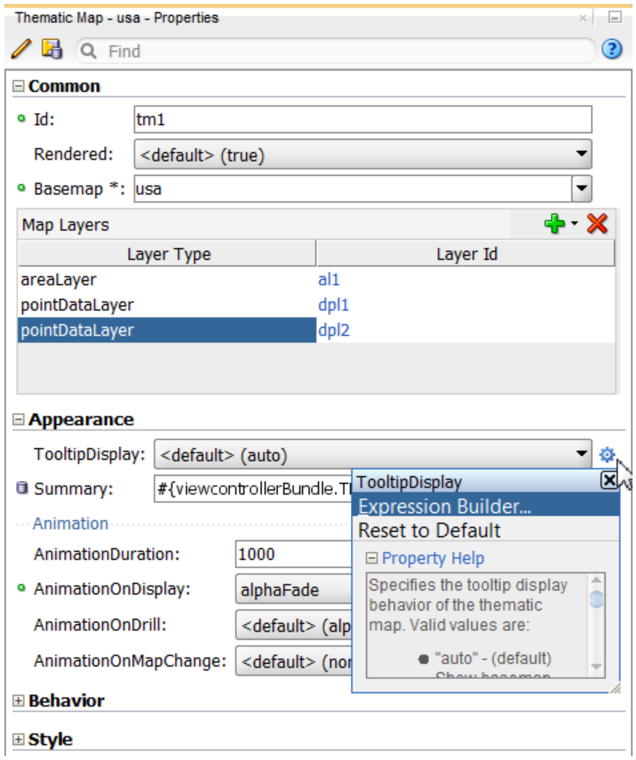
Figure 28–25 Component Gallery for Thematic Maps



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 28–26 shows the dropdown menu for a thematic map component `tooloTipDisplay` attribute.

Figure 28–26 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. For more information, 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. For more information, see Section 28.6.1, "Configuring Thematic Maps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 28.1.3, "Additional Functionality for Map Components."

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.

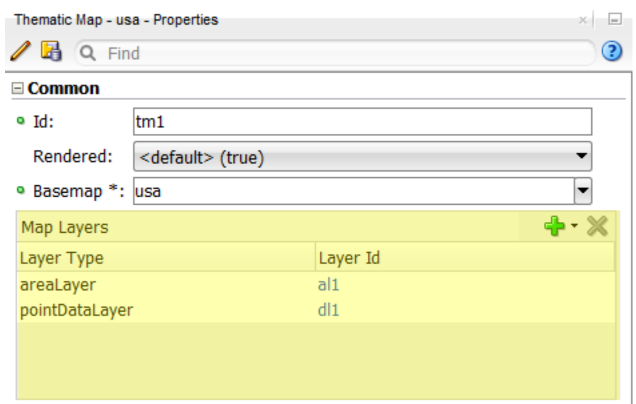
   Use the dialog to select the thematic map base map and layer in the prebuilt map hierarchy that you want the thematic map to display. For help with the dialog, press F1 or click **Help**.

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**, **europe**, **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 28–27 shows the map layers dialog highlighted in the Properties window for the `thematicMap` component.

Figure 28–27 Map Layers Dialog in the Properties Window



Alternatively, you can use the Layer Browser to add map layers to the thematic map. For more information, see Section 28.6.3, "Using the Layer Browser."

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. For more information, see Section 28.7, "Customizing Thematic Map Display Attributes."
   - **Animation** subsection: Use the animation attributes in this subsection to configure animation in thematic maps. For more information, see Section 28.8.3, "How to Configure Animation Effects."
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**.

6. Expand the **Other** category. For the **Summary** attribute, enter a description of the thematic map. This description is accessed by screen reader users.

28.6.5 What Happens When You Add a Thematic Map to a Page

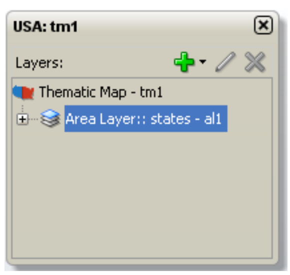
When you use the Components window to create a thematic map, JDeveloper inserts code in the JSF page. Example 28–7 shows the code inserted in the JSF page.

Example 28–7 Thematic Map Code In JSF Page

```
<dvt:thematicMap basemap="usa" id="tm1">
  <dvt:areaLayer layer="states" id="al1"/>
</dvt:thematicMap>
```

The Layer Browser displays the hierarchical structure of the thematic map. Figure 28–28 shows the Layer Browser after using the Components window to create a thematic map.

Figure 28–28 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*.

28.6.6 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 Section A.4.3, "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

28.7 Customizing Thematic Map Display Attributes

You can customize the display attributes of 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. Data values that require special formatting, for example, currency or percentages, can be configured to display special symbols and decimal points. You can also configure tooltips to display data when the user moves the cursor over the map.

28.7.1 How to Customize Thematic Map Labels

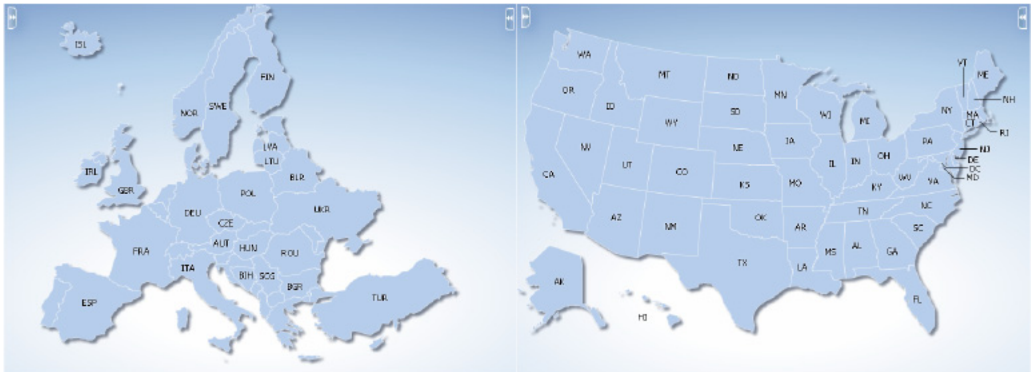
By default, each region in each map layer for a prebuilt base map has a label with a short and long type, for example, `BRA` and `Brazil` in the `countries` layer of the `world` base map. The map layer is specified by the `areaLayer` component in the `layer` attribute, for example:

```
<dvt:areaLayer id="al1" 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 28–29 shows the default labels for the `europe` and `usa` base maps.

Figure 28–29 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 Section 28.6.1, "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 Section 28.6.4, "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 Section 28.7.2, "How to Configure Tooltips to Display Data."

28.7.2 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 Section 28.6.1, "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 Section 28.6.4, "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.

28.7.3 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 Section 28.6.1, "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 Section 28.6.4, "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 Section 28.7.1, "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.

Example 28–8 shows sample code for formatting numeric data values for an area and a marker label.

Example 28–8 Sample Code for Area and Marker Numeric Data Value Formatting

```
...
<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}"/>
```

28.8 Adding Interactive Features to Thematic Maps

Thematic maps include support for interaction features including selection and action events, drilling, popups, animation, and drag and drop operations.

28.8.1 How to Configure Selection and Action Events in Thematic Maps

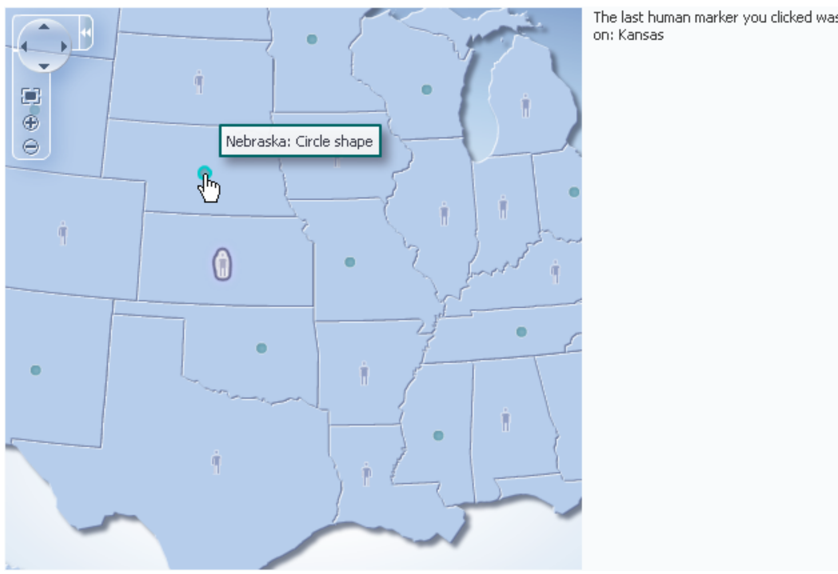
You can configure your thematic map components to allow users to select one or more areas or markers across multiple data layers. By default, selection is not enabled.

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 Chapter 6, "Handling Events."

Figure 28–30 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 28–30 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 Section 28.6.1, "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 Section 28.6.4, "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}`

shows sample code for configuring markers to fire action events.

Example 28–9 Sample Code for Thematic Map Marker 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*.

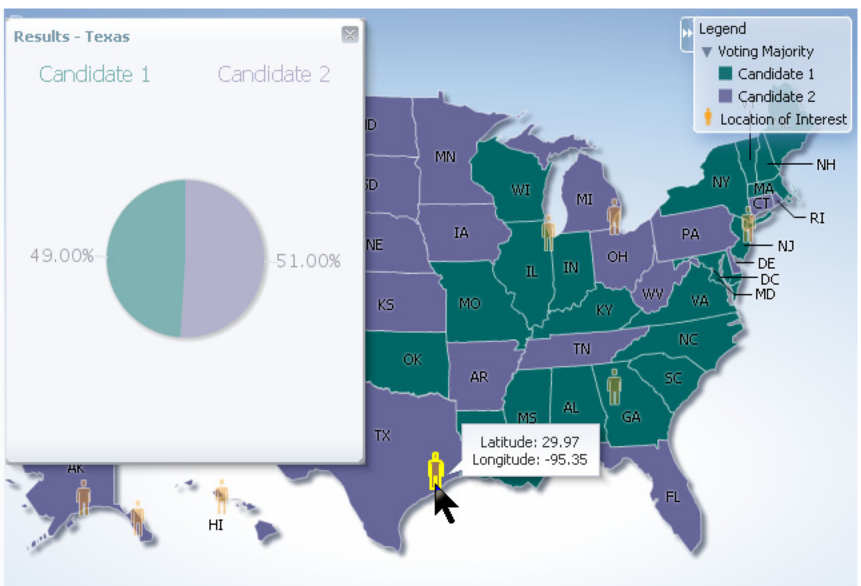
28.8.2 How to Add Popups to Thematic Map Areas and Markers

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 Chapter 15, "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 28–31 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 28–31 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 Section 28.6.1, "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 Section 28.6.4, "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. Example 28–10 shows sample code for the dialog to be referenced when an area stamp is clicked.

Example 28–10 Code Sample for the Area Dialog Popup

```
<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="pgl6">
      <af:panelGroupLayout id="pgl7" 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="pgl5" 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>
```

Example 28–11 shows sample code for the note window to be referenced when the user hovers the mouse over a marker stamp.

Example 28–11 Sample Code for Marker Note Window

```
<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="pgl8" 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 Chapter 15, "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.

Example 28–12 shows sample code for adding popup components to the area and marker stamps in a thematic map.

Example 28–12 Popups Associated With Area and Marker Components

```
<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:area1"/>
    </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="area1"
                 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="area1"
                 align="endAfter"/>
      </dvt:area>
    </dvt:areaLocation>
  </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>
```

28.8.3 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 Section 28.6.1, "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 Section 28.6.4, "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 28–2 shows the animation effect available for each supported thematic map event.

Table 28–2 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

28.8.4 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 Section 28.6.1, "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 Section 28.6.4, "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.

Example 28–13 shows sample code for adding and configuring an area as a drag source.

Example 28–13 Sample Code for 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.

Example 28–14 shows sample code for adding and configuring a map layer as a drop target.

Example 28–14 Sample Code for Area 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 Section 36.5, "Adding Drag and Drop Functionality for Components."

29

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:

29.1 About Hierarchy Viewer Components

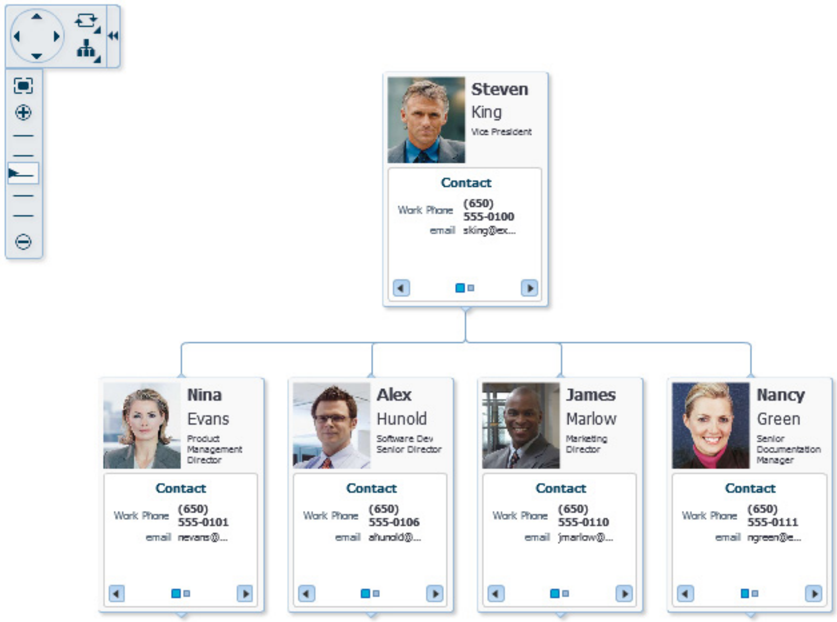
Hierarchy viewers are used to display hierarchical data. Hierarchical data contains master-detail relationships within the data. For example, you could create a hierarchy viewer that renders an organization chart from a data collection that contains information about the relationships between employees 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.

29.1.1 Hierarchy Viewer Use Cases and Examples

A hierarchy viewer visually displays hierarchical data and the master-detail relationships. Figure 29–1 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 29–1 Hierarchy Viewer Component with Control Panel and Nodes



29.1.2 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.

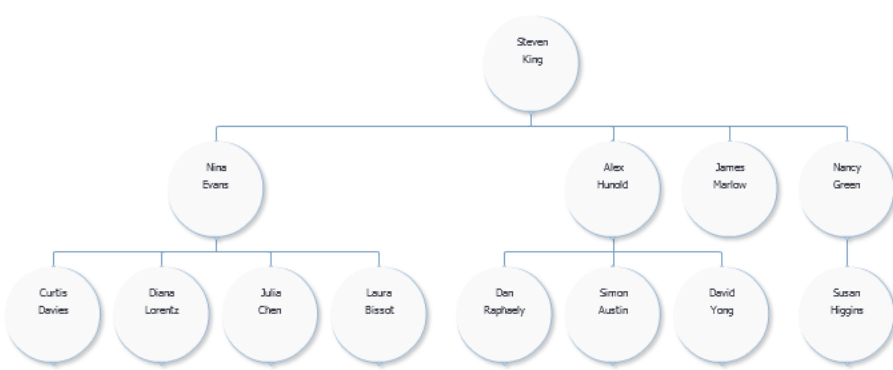
29.1.2.1 Layouts

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 29–2 shows an example of a vertical top down layout.

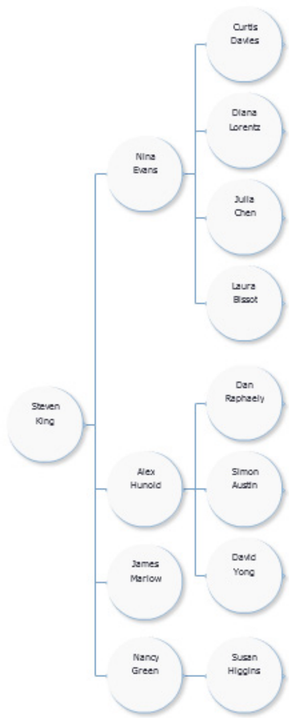
Figure 29–2 Hierarchy Viewer Vertical Top Down Layout



- Vertical bottom up
- Horizontal left-to-right

  Figure 29–3 shows an example of a horizontal left-to-right layout.

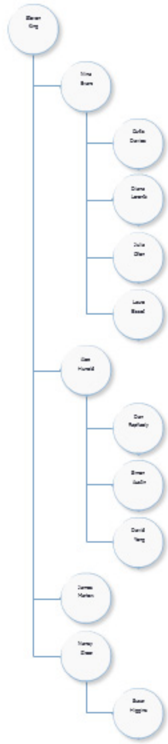
Figure 29–3 Hierarchy Viewer Horizontal Left-to-Right Layout



- Horizontal right-to-left
- Horizontal, direction depends on the locale
- Tree, indented tree

  Figure 29–4 shows an example of a tree layout.

Figure 29–4 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 29–5 shows an example of a circle layout.

Figure 29–5 Hierarchy Viewer Circle Layout

29.1.2.2 Navigation

At runtime, the node contains controls that allow users to navigate between nodes and to show or hide other nodes by default.

At runtime, the end user uses the controls on the node to switch dynamically between the content that the panel cards reference.

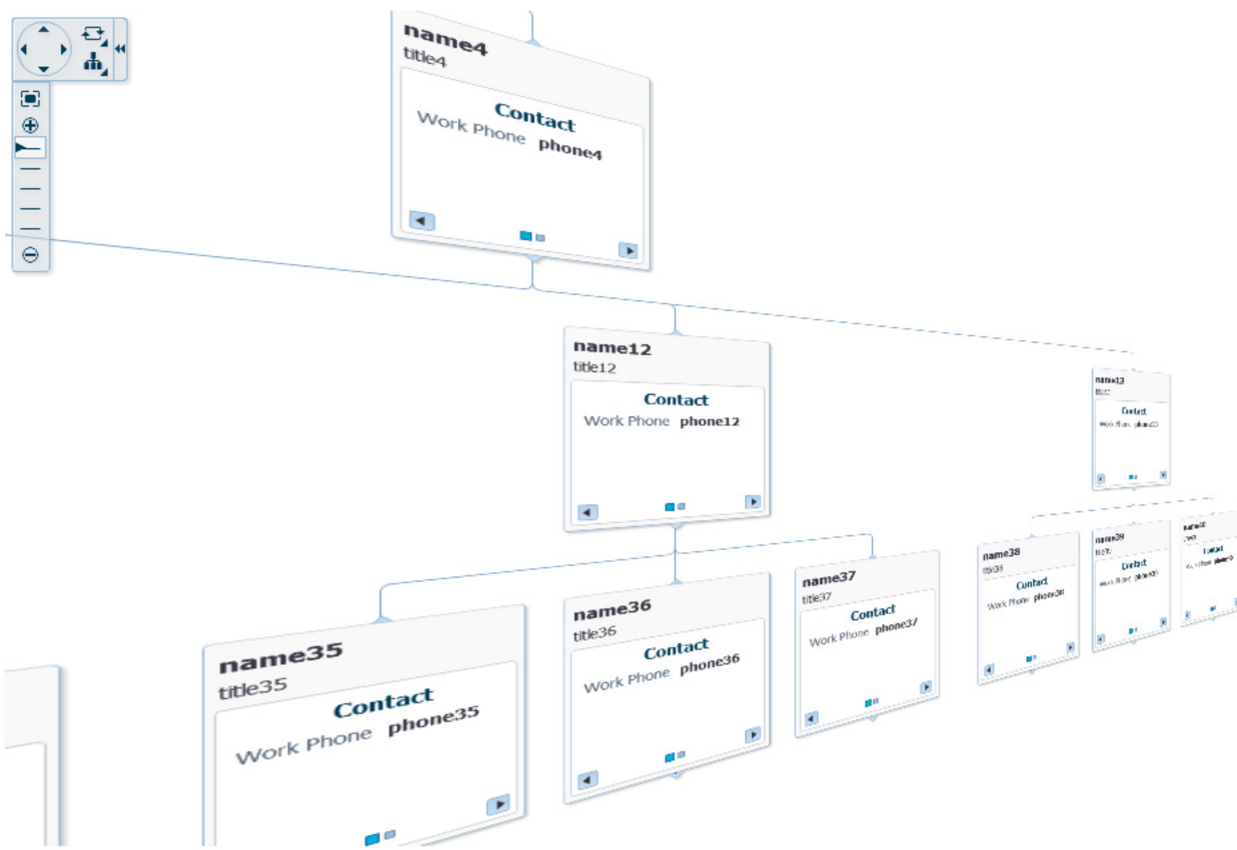
At runtime, if a user double-clicks another node that has a value specified for its `setAnchorListener` property, that node becomes the anchor node.

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.

29.1.2.3 Tilt Panning

If enabled, instead of browsing through a hierarchy viewer with a large quantity of nodes one page at a time, users can initiate a tilt panning effect that animates the hierarchy viewer to fly visually through the hierarchy viewer nodes. Once set in motion toward the edge of a view, the effect continues automatically until it reaches the end of the nodes on an edge. Figure 29–6 shows the tilt panning effect as it reaches the edge of the view.

Figure 29–6 Hierarchy Viewer Tilt Panning Effect



To use the tilt panning effect you should first adjust the zoom level on the hierarchy view for an acceptable view of the content of the nodes. You can initiate the effect in any of these ways:

- Click and drag when using the pan control in the control panel to initiate tilt panning after a short period of regular panning.
- Click and drag the view one-third of the way across the viewport.
- Click and hold the cursor near the edge of the view to initiate tilt panning in that direction.

Once the tilt panning effect is initiated, you can move the mouse within the view to change the direction of the pan through the view. To exit tilt panning, release the mouse button.

29.1.2.4 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 29–7.

Figure 29–7 Control Panel in Hidden State

Users click the **Hide or Show Control Panel** button shown in Figure 29–7 to hide or expand the Control Panel. Figure 29–8 shows the expanded Control Panel.

Figure 29–8 Control Panel in Show State

Table 29–1 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 29–1 Elements in the Control Panel

Control	Name	Description
	Pan Control	Allows user to reposition the hierarchy viewer component within the viewport.
	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 `af:showDetailItem` elements that you have defined. Users can use the panel selector to show the same panel on all nodes at once.
	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.

29.1.2.5 Printing

Hierarchy viewers are printed using the HTML view in the browser.

29.1.2.6 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.

29.1.2.7 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`.

29.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- Accessibility: You can make your hierarchy viewer components accessible. For more information, see Chapter 33, "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 Section 12.2.2, "Content Delivery."
- 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 Appendix D, "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 Section 22.2, "Common Functionality in Data Visualization Components."

29.2 Using Hierarchy Viewer Components

A hierarchy viewer component requires data collections where a master-detail relationship exists between one or more detail collections and a master detail collection. The hierarchy viewer component uses the same data model as the ADF Faces `tree` component. 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 the "Using Trees to Display Master-Detail Objects" section 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 more information about navigating a hierarchy viewer component, see Section 29.5, "Configuring Navigation in a Hierarchy Viewer."

29.2.1 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 Section 29.3.1, "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 Section 29.6.4, "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.

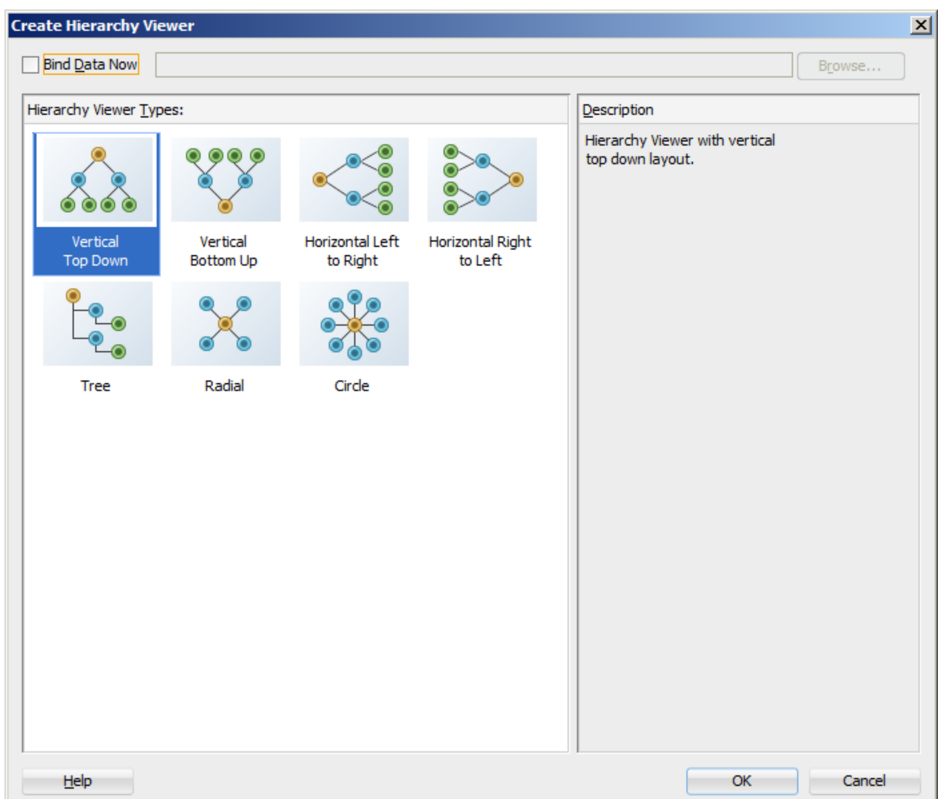
  The `f:facet` tag for each zoom level supports the use of a `dvt:panelCard` element that contains one or more `af:showDetailItem` elements defining the content to be displayed at the specified zoom level. At runtime, the end user uses the controls on the node to switch dynamically between the content that the `af:showDetailItem` elements reference. For more information, see Section 29.4, "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.

29.2.2 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 29–9 shows the Create Hierarchy Viewer dialog for hierarchy viewers with the vertical top down layout type selected.

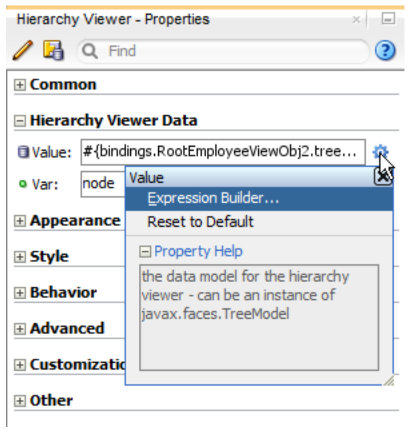
Figure 29–9 Create Hierarchy Viewer Dialog



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 29–10 shows the dropdown menu for a hierarchy viewer component `Value` attribute.

Figure 29–10 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 Section 29.2.1, "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 Section 29.1.3, "Additional Functionality for Hierarchy Viewer Components."

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 Section 29.1.2.1, "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 Section 29.6.3, "How to Configure the Display of the Control Panel."

   - **Panning**: Specify panning behavior. The default value is `default` for click and drag panning. You can also specify a `tilt` value for click and drag panning with automatic 3D tilt panning enabled.

29.2.3 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. Example 29–1 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.

Example 29–1 Hierarchy Viewer Sample Code

```
<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: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 AFHVNodeStretchHeight
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>
```

29.2.4 What You May Need to Know About Hierarchy Viewer Rendering and HTML

By default, the hierarchy viewer component renders in a Flash Player. When Flash 10 or higher is not available on the client or for the purpose of printing, the hierarchy viewer is rendered in HTML. While HTML rendering follows 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.

29.3 Managing Nodes in a Hierarchy Viewer

A node is a shape that represents the individual elements in a 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.

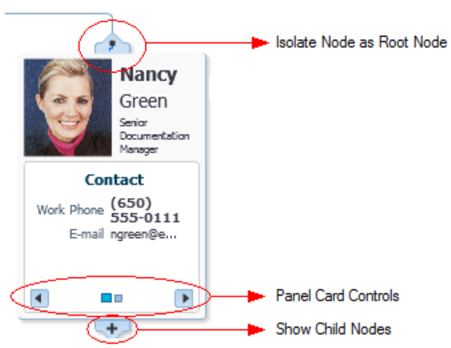
- Show or hide child nodes of the currently selected node in a hierarchy viewer component.

  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 Section 12.6.4, "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 Section 29.4, "Using Panel Cards."

Figure 29–11 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 Section 29.3.2, "How to Configure the Controls on a Node."

Figure 29–11 Hierarchy Viewer Node Controls



29.3.1 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 Chapter 9, "Organizing Content on Web Pages."
- Menu components including: `af:menu` and `af:menuItem`. For more information about these components, see Chapter 16, "Using Menus, Toolbars, and Toolboxes."
- Output components including: `af:outputFormatted` and `af:outputText`. For more information about these components, see Chapter 18, "Using Output Components."
- Message component `af:panelLabelAndMessage`. For more information about this component, see Chapter 19, "Displaying Tips, Messages, and Help."
- Navigation components including: `af:button`, `af:link`, and `af:commandMenuItem`. For more information about these components, see Chapter 20, "Working with Navigation Components."
- Image component `af:image`. For information about how to use the `af:image` component, see Section 29.6.2, "Including Images in a Hierarchy Viewer."
- `af:showPopupBehavior:` For information about how to use the `af:showPopupBehavior` component, see Section 29.7.2, "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 Section 29.4, "Using Panel Cards."

Note: Unsupported components are flagged at design time.

By default, the hierarchy viewer component renders in a Flash Player and in HTML for the purpose of printing. For this reason, certain properties of ADF Faces components that you specify as node content may not be supported, and the component may not render as you expect. For more information, see Section 29.2.4, "What You May Need to Know About Hierarchy Viewer Rendering and HTML."

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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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 Section 29.3.3, "Specifying a Node Definition for an Accessor."

29.3.2 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.3.1, "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 29–2.

Table 29–2 Node Configuration Properties

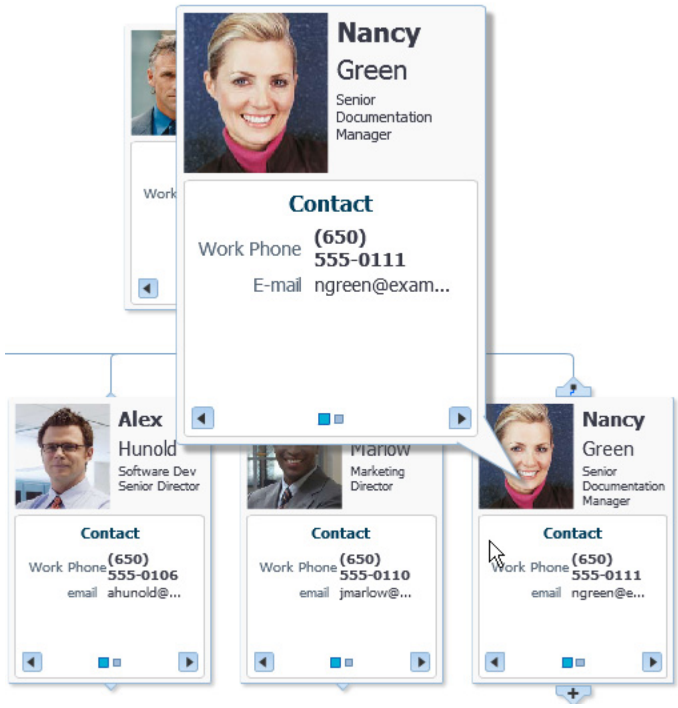
To do this:	Set the following value for this property:
Configure the Hide or Show controls to appear or not on a node.	Set `showExpandChildren` to `False` to hide the controls. By default the property is set to `True`.
Configure the Restore or Isolate controls to appear or not on the node.	Set the `showIsolate` property to `False` to hide these controls on the node. By default the property is set to `true`.
Configure the Navigate Up control to appear or not on the node.	Set the `showNavigateUp` property to `False` to hide this control on the node. By default the property is set to `true`. If the `showNavigateUp` property is set to `true`, for the control to render, you must also set a value for the hierarchy viewer component's `navigateUpListener` property, as described in Section 29.5.1, "How to Configure Upward Navigation in a Hierarchy Viewer."
Configure the height and width of a node.	Set values for the `width` and `height` properties.
Select the shape of the node.	Select a value from the **Shape** dropdown list. Available values are: ▪ `ellipse` ▪ `rect` ▪ `roundedRect` (default)

3. For information about configuring the properties in the Style section of the Properties window for the node component, see Section 31.2, "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 29–12.

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 Section 29.6.5, "How to Disable the Hover Detail Window."

Figure 29–12 Hover Window in Hierarchy Viewer Node



29.3.3 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 Section 29.3.4, "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.

29.3.4 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.

29.3.5 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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.

29.4 Using Panel Cards

You can use the panel card component in conjunction with the 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.

29.4.1 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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 Section 9.13.1, "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.

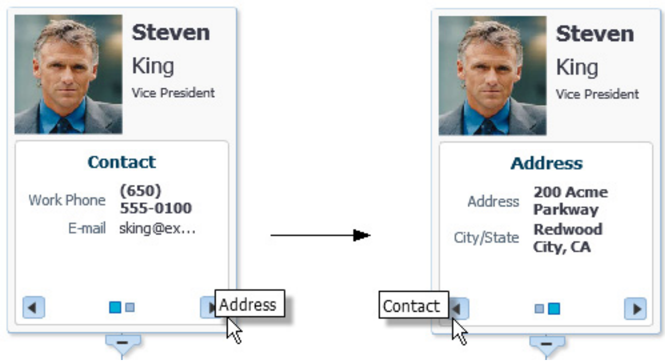
   For example, if your hierarchy viewer renders an organization chart, you could add an element to display the employee's ID. In the Structure window, right-click **af:showDetailItem** and choose **Insert Inside Show Detail Item > ADF Faces > Output Text**.
9. Use the Properties window to configure the 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.

29.4.2 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 29–13 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 29–13 Runtime View of a Node with a Panel Card



29.5 Configuring Navigation in a Hierarchy Viewer

By default, a 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 Section 29.3, "Managing Nodes in a Hierarchy Viewer."

29.5.1 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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.

29.5.2 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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.

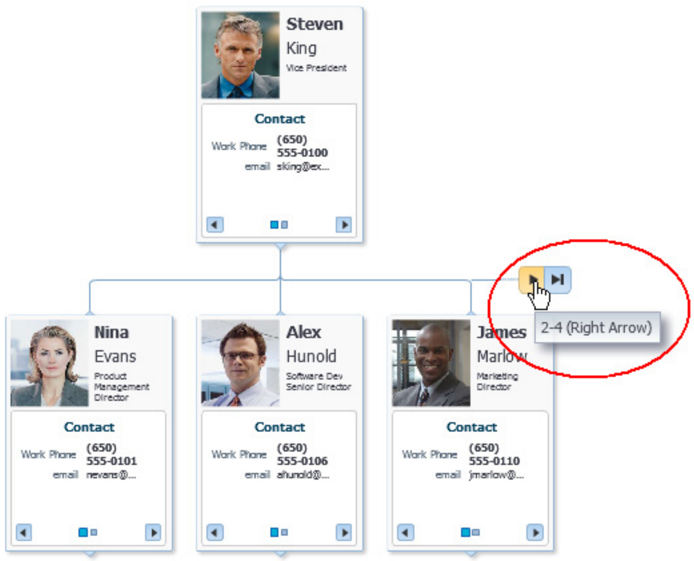
29.5.3 What Happens When You Configure Same-Level Navigation in a Hierarchy Viewer

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 29–14 shows a runtime example where `levelFetchSize="3"`. When a user moves the mouse over the control, as shown in the circled area in Figure 29–14, the control that allows users to navigate to the last set of nodes appears.

Figure 29–14 Hierarchy Viewer Component with Same-Level Navigation



29.6 Customizing the Appearance of a Hierarchy Viewer

You can customize the hierarchy viewer component size and appearance including adding images, configuring the display of the control panel, 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 Chapter 31, "Customizing the Appearance Using Styles and Skins."

29.6.1 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 Section 31.2, "Changing the Style Properties of a Component."

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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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.

29.6.1.1 What You May Need to Know About Skinning and the hierarchyViewer Component

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.

Example 29–2 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.

Example 29–2 Using a Skinning Key to Change Panel Card Navigation Button When Selected

```
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 skinning and styles, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

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 Chapter 9, "Organizing Content on Web Pages."

29.6.2 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 Section 9.13.1, "How to Use the panelGroupLayout Component."

29.6.3 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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`

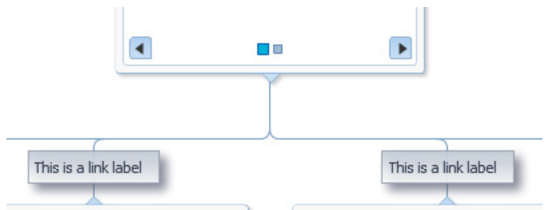
  Select this value if you want the Control Panel to appear in an expanded or show state at runtime.

3. Save changes to the JSF page.

29.6.4 How to Configure the Display of Links and Labels

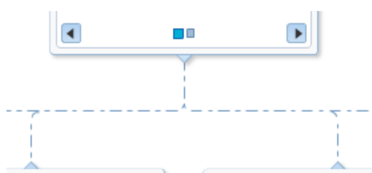
In a hierarchy viewer the relationships between nodes are represented by lines that link the nodes. The links can be configured to include labels. Figure 29–15 illustrates links and labels in a hierarchy viewer.

Figure 29–15 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 29–16 illustrates links with a `dashDot` value set for the `linkStyle` attribute.

Figure 29–16 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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 Section 3.5, "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"}
   ```

29.6.5 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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:
   - `default`

     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.

29.7 Adding Interactivity to a Hierarchy Viewer Component

You can configure a 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.

29.7.1 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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.

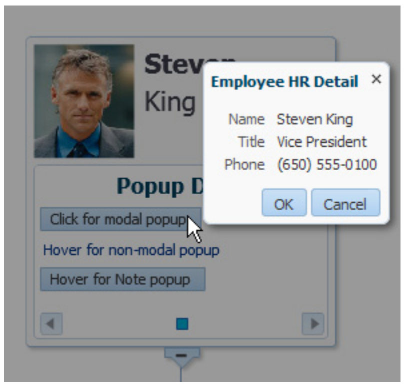
29.7.2 Configuring a Hierarchy Viewer to Invoke a Popup Window

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 29–17 shows a modal popup invoked from an **HR Detail** link in the node. Example 29–3 shows sample code for creating the popup.

Figure 29–17 Modal Popup in Hierarchy Viewer Node



Example 29–3 Sample Code to Create the Popup

```
<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="Offical Title" >
        <af:outputText  value="#{myBean.selectedEmployee.officalTitle}"/>
      </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>
```

Example 29–4 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.

Example 29–4   Sample Code to Invoke Popup from Hierarchy Viewer Component

```
<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 = bu1>
        <af:showPopupBehavior popupId="::popupDialog" triggerType="action"
                             align="endAfter" alignId="bu1" />
      </af:button>
    </showDetailItem>
  </dvt:panelCard>
</f:facet>
```

For more information about using the `af:showPopupBehavior` tag, see Section 15.3, "Declaratively Invoking a Popup."

29.7.3 Configuring Hierarchy Viewer Drag and Drop

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 29–18 shows a hierarchy viewer configured to allow drags and drops within itself. In this example, 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 29–18   Hierarchy Viewer Showing a Node Drag

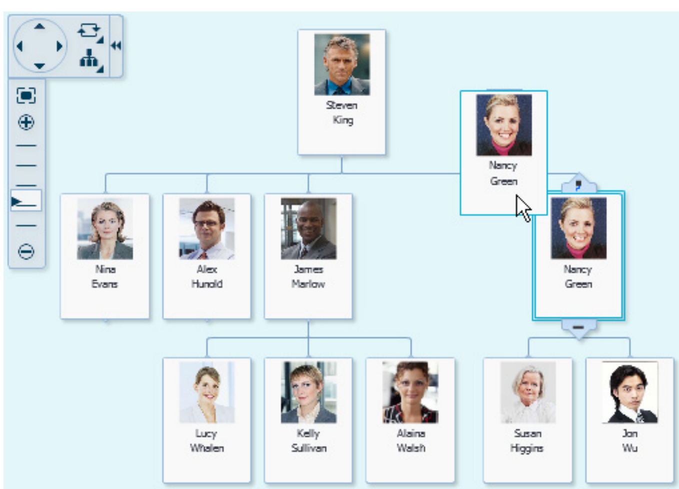
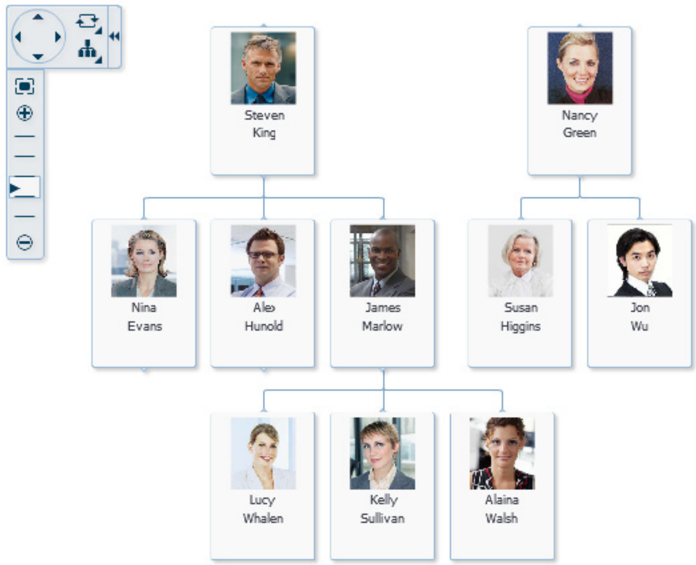


Figure 29–19 shows the result of a drag to the hierarchy viewer background. Nancy Green and her subordinates are now shown as a new tree in the hierarchy.

Figure 29–19 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 29–20 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 29–20 Hierarchy Viewer After Node Drag to Another Node

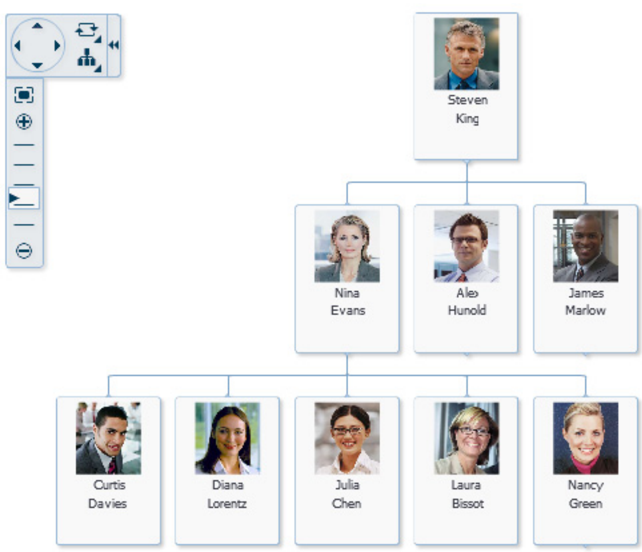


Figure 29–21 shows an example of the same hierarchy viewer configured to allow drops to or drags from an `af:outputFormatted` component. In this example, the user can drag one or more nodes to the drop text, and the text will change to indicate which node(s) the user dragged and which operation was performed. 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 29–21 Hierarchy Viewer Configured for Drag and Drop to Another Component

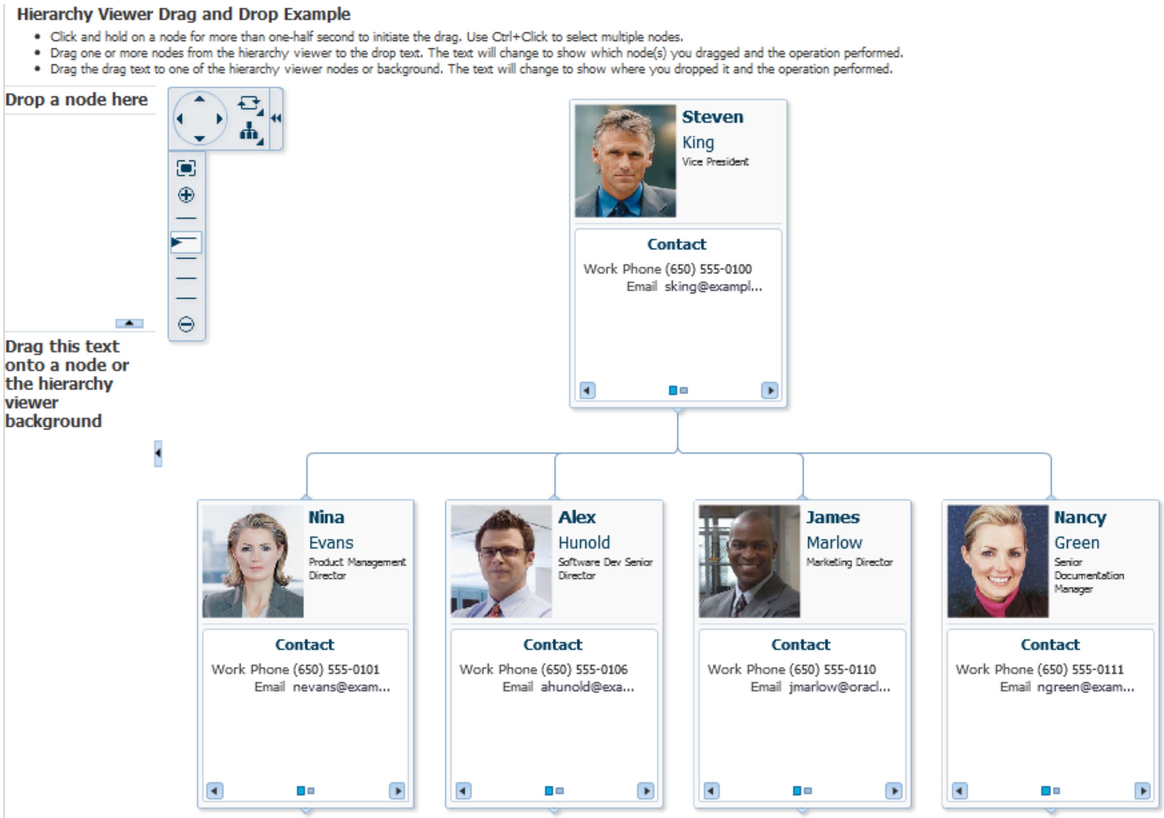
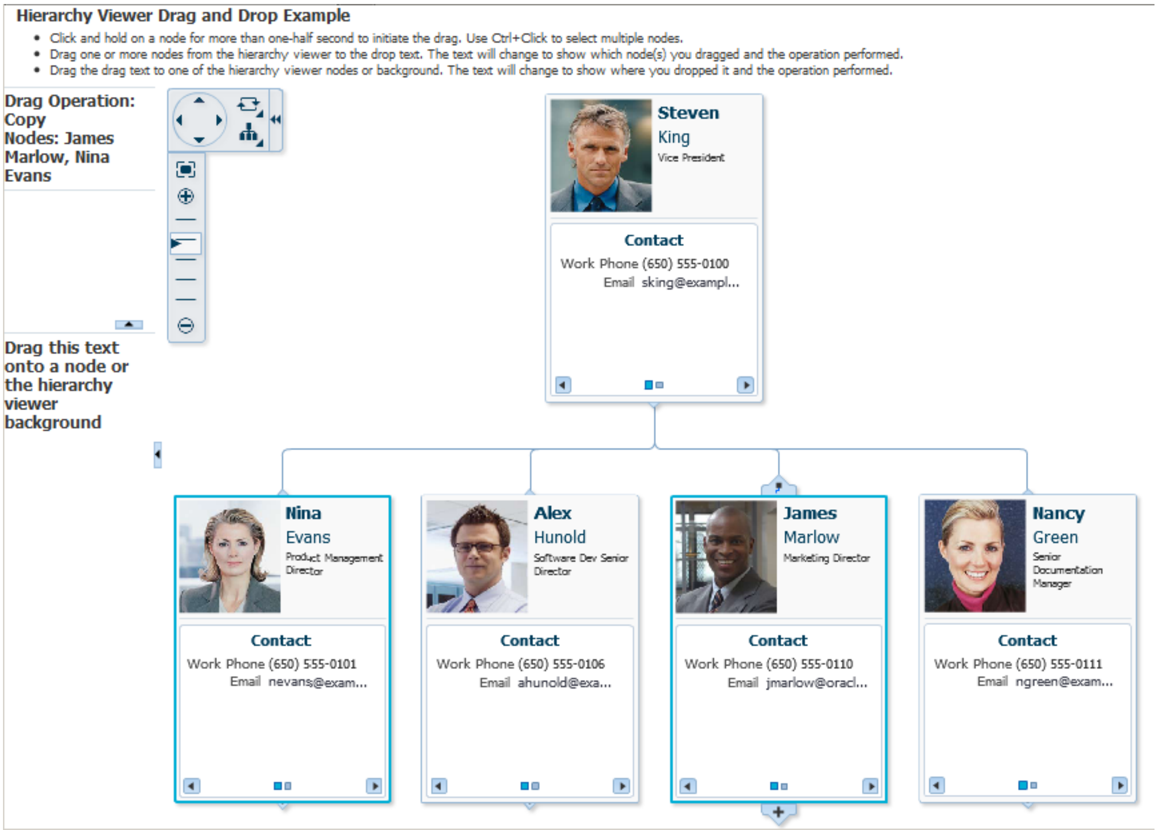


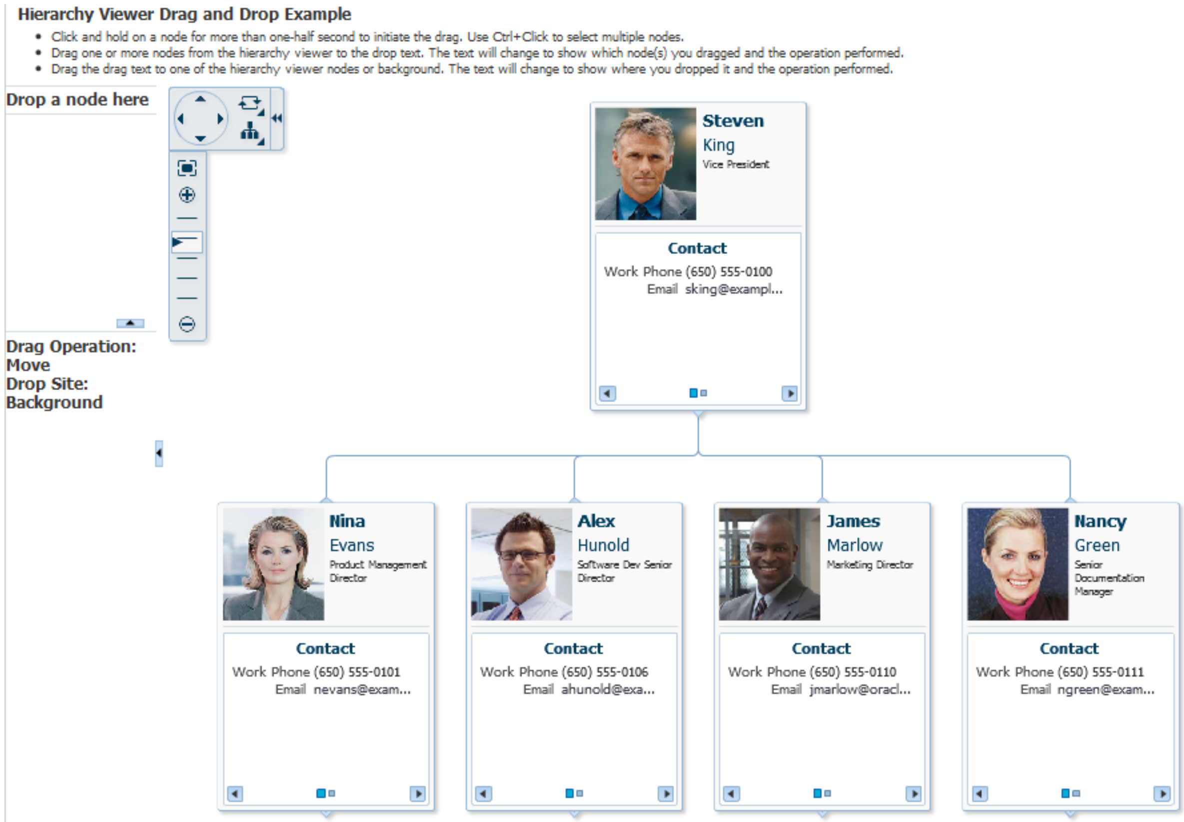
Figure 29–22 shows the same hierarchy viewer after the user dragged the nodes containing the data for Nina Evans and James Marlow to the drop text.

Figure 29–22 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 29–23 shows the same hierarchy viewer after a user drags the text to the hierarchy viewer background.

Figure 29–23 Hierarchy Viewer After Text Drag to Hierarchy Viewer Background



29.7.3.1 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 29–21. For more detailed information about configuring drag and drop on ADF Faces or ADF Data Visualization components, see Chapter 36, "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 Section 29.2.1, "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 Section 29.1.3, "Additional Functionality for Hierarchy Viewer Components."

You will need to complete these tasks:

- Add a hierarchy viewer to your page. For more information, see Section 29.2.2, "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 29–21 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.

  Example 29–5 shows the completed page for the additional components. The hierarchy viewer details are omitted.

Example 29–5 Sample Code for Hierarchy Viewer Drag and Drop Example

```
<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="pl1"
                      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 Chapter 18, "Using Output Components." For help with the `af:panelGroupLayout` component or other page layout components, see Chapter 9, "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.

Example 29–6 shows the sample drop listener and supporting methods for the hierarchy viewer displayed in Figure 29–21.

Example 29–6 Sample Drop Listener for a Hierarchy Viewer

```
// 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.getDragComponent());
  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 demo application. For more information about the demo application, see Chapter 2, "ADF Faces 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:
   1. 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.

2. In the Structure window, right-click the **af:dragSource** node and choose **Go to Properties**.

3. 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 29–21 uses `MOVE` as the default action.

11. To make another component the drop target for drags from the hierarchy viewer, do the following:

    1. 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 29–21 contains an `af:outputFormatted` component that displays the results of the drop.

    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 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.

       Example 29–7 shows the sample drop listener for the hierarchy viewer displayed in Figure 29–21. This example uses the same imports and helper methods used in, and they are not included here.

Example 29–7 Sample Drop Listener for a Component Using a Hierarchy Viewer as a Drag Source

```
// 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();
  }
```

3. Click **OK** to enter the Insert Data Flavor dialog.
4. 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.
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. 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}
   ```

29.7.3.2 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`.

29.8 Adding Search to a Hierarchy Viewer

The hierarchy viewer 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 29–24 shows a sample search panel.

Figure 29–24 Hierarchy Viewer Search Panel

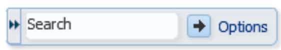


Figure 29–25 shows sample search results.

Figure 29–25 Hierarchy Viewer Sample Search Results

29.8.1 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 Section 29.2.1, "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 Section 29.1.3, "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 Section 29.2.2, "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:
   1. In the Structure window, expand the **dvt:search** node.
   2. Right-click the **af:commandLink** node and choose **Go to Properties**.
   3. In the Properties window, configure any desired properties for the **af:commandLink** node.

      For example, to change the text that displays `Advanced` for the advanced search, enter a value for the **Text** attribute.
   4. To add a popup that will display the advanced search options to the user, in the Structure window, right-click the **af:commandLink** node and choose **Insert Inside Command Link > ADF Faces > Show Popup Behavior**.
   5. 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 Section 15.3, "Declaratively Invoking a Popup." For more information about the `query` component, see Section 14.4, "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.

Example 29–8 shows sample code for configuring search in a hierarchy viewer.

Example 29–8 Sample Hierarchy Viewer Search Code

```
<dvt:hierarchyViewer>
  ... hierarchy viewer details omitted
  <dvt:search id="searchId" value="#{bindings.lastNameParam.inputValue}"
              actionListener="#{bindings.ExecuteWithParams1.execute}">
    <f:facet name="end">
      <af:commandLink text="Advanced">
        <af:showPopupBehavior popupId="::mypop" triggerType="action"/>
      </af:commandLink>
    </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>
```

29.8.2 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 Section 9.13.1, "How to Use the panelGroupLayout Component."

30 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:

30.1 About the Treemap and Sunburst Components

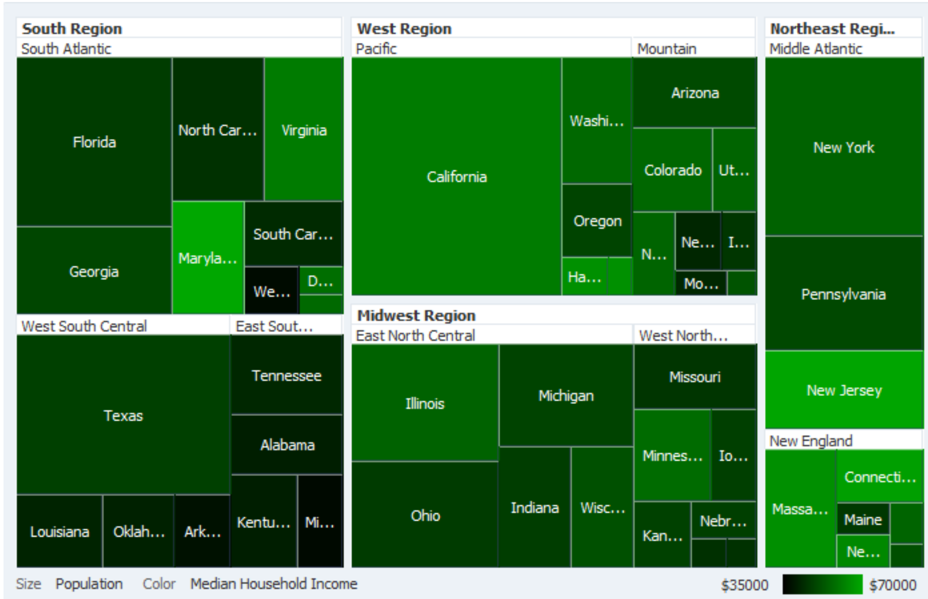
Use *treemaps* and *sunbursts* to display quantitative hierarchical data across two dimensions, represented visually by size and color. Treemaps and sunbursts use a shape called a `node` to reference the data in the hierarchy.

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. The appearance and content of the nodes are configurable at each level of the hierarchy.

30.1.1 Treemap and Sunburst Use Cases and Examples

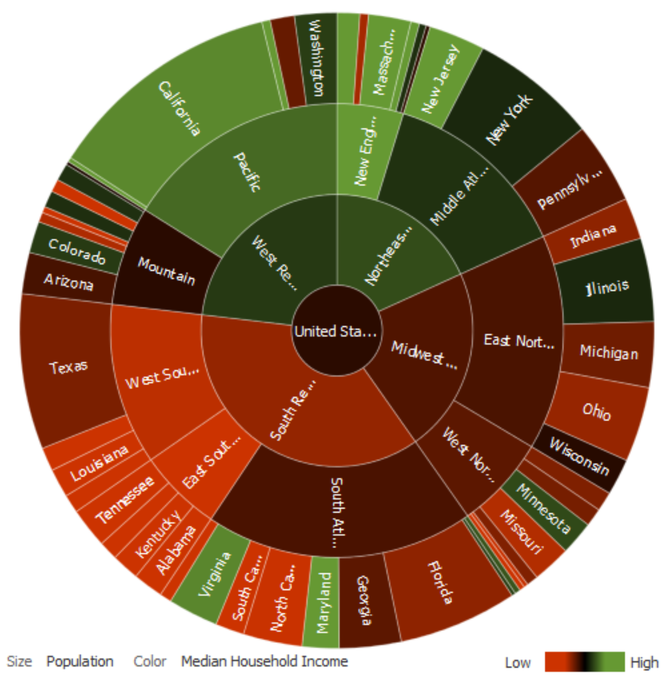
Treemaps display nodes as a set of nested rectangles. Each branch of the tree is given a rectangle, which is then tiled with smaller rectangles representing sub-branches. Figure 30–1 shows a treemap displaying United States census data grouped by regions, with the color attribute used to indicate median income levels. States with larger populations display in larger-sized nodes than states with smaller populations.

Figure 30–1 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 30–2 shows the same census data displayed in a sunburst.

Figure 30–2 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 Chapter 12, "Using Tables, Trees, and Other Collection-Based Components." For information about using hierarchy viewers, see Chapter 29, "Using Hierarchy Viewer Components."

30.1.2 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:

30.1.2.1 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 30–2.

The layout of nodes in a `treemap` component is configurable and includes the following types of layouts:

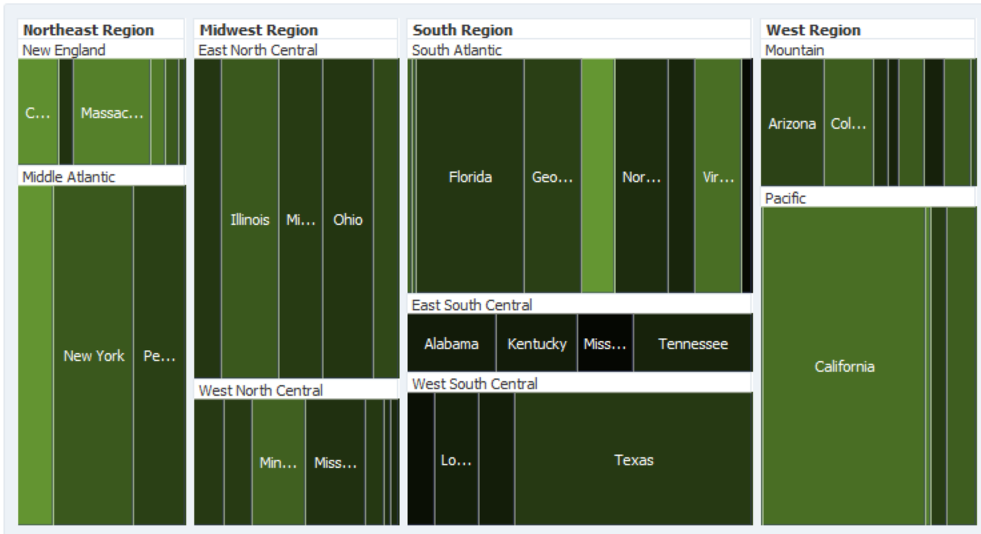
- Squarified, nodes are laid out to be as square as possible.

  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 30–1.

- 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 30–3 displays the sample United States census data in the slice and dice horizontal layout.

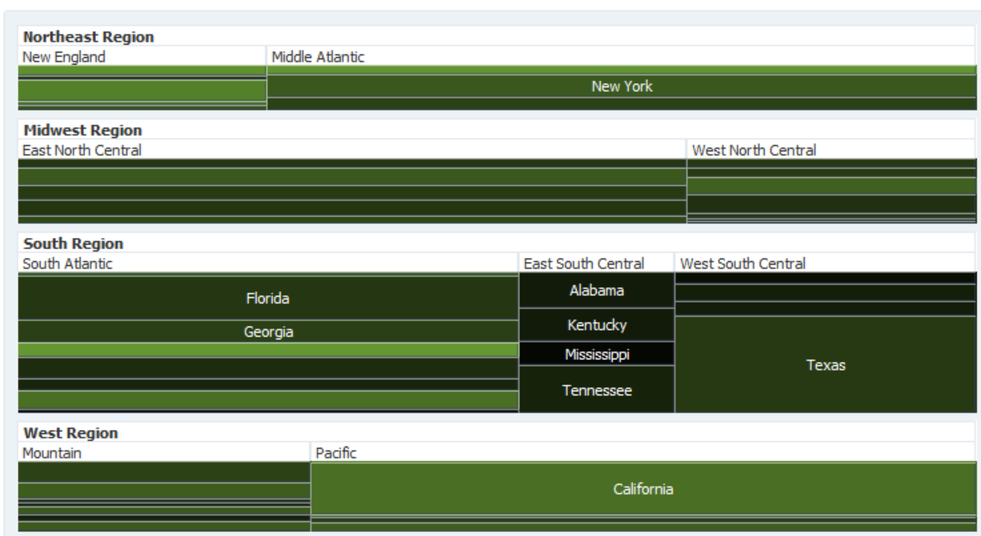
Figure 30–3 Treemap Displaying United States Census Data in Slice and Dice Horizontal Layout



- 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 30–4 displays the sample United States census data in the slice and dice vertical layout.

Figure 30–4 Treemap Displaying United States Census Data in Slice and Dice Vertical Layout



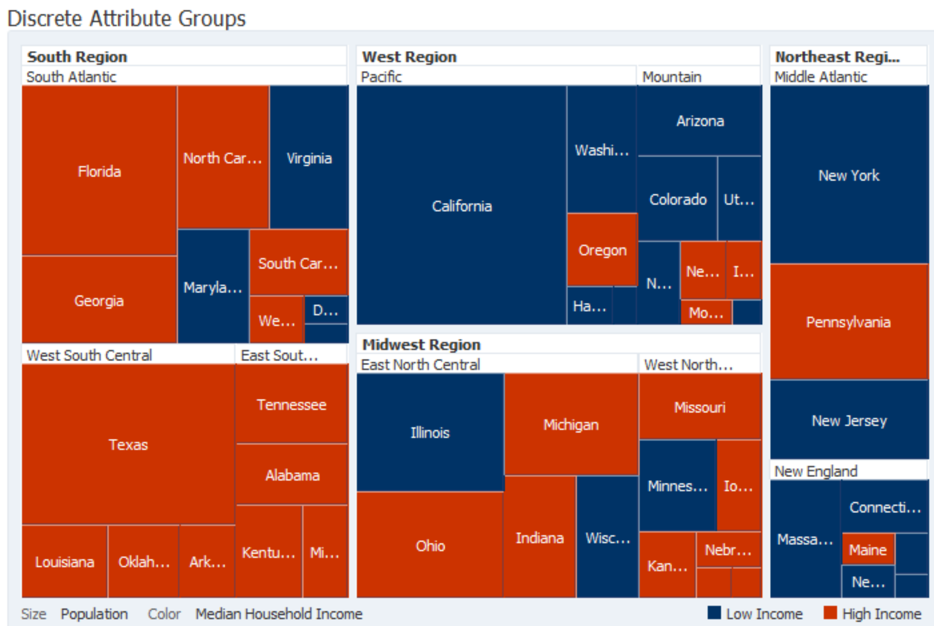
30.1.2.2 Attribute Groups

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 30–1 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 30–5 shows the same United States census population using two colors to distinguish between high and low median incomes.

Figure 30–5 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.

30.1.2.3 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 30–5 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.

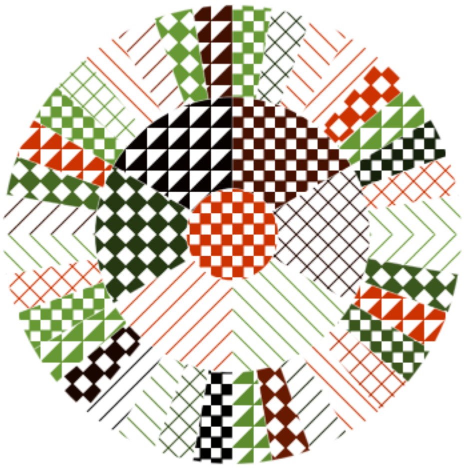
If your treemap uses continuous attribute groups, the legend displays the colors used as a gradient. The treemap in Figure 30–1 shows a legend for a treemap using continuous attribute groups to indicate median income levels.

30.1.2.4 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 30–6 shows a sunburst configured with an assortment of fill patterns.

Figure 30–6 Sunburst Illustrating Pattern Fill

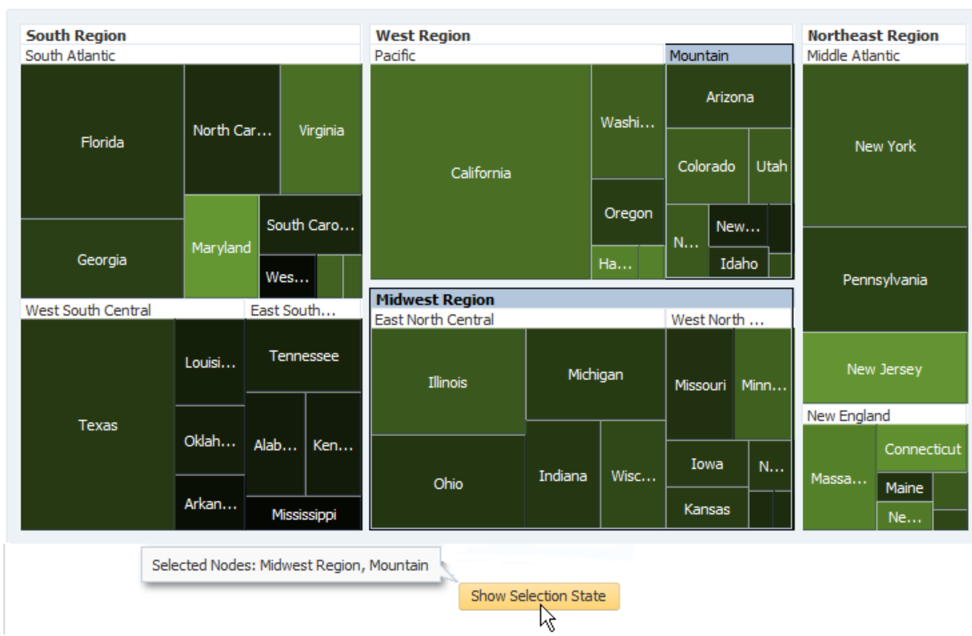


30.1.2.5 Node Selection Support

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 30–7 shows a treemap configured for multiple selection support.

Figure 30–7 Treemap Illustrating Multiple Selection Support

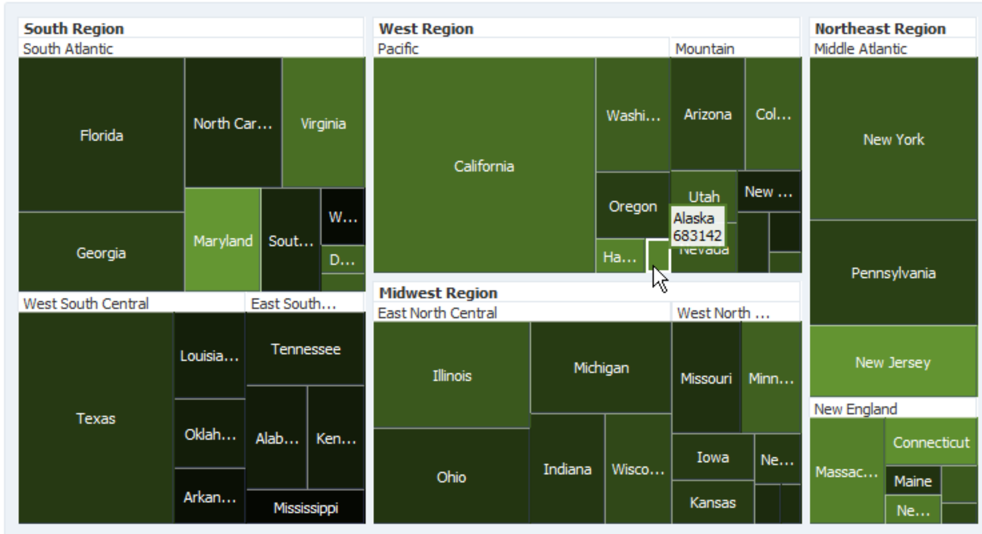


30.1.2.6 Tooltip Support

Treemaps and sunbursts support the ability to display additional information about a node when the user moves the mouse over a node.

Figure 30–8 shows the tooltip that is displayed when the user moves the mouse over the Alaska node.

Figure 30–8 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.

30.1.2.7 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 30–9 shows a sample popup displayed when a user hovers the mouse over one of the treemap nodes.

Figure 30–9 Treemap Popup on Mouse Hover

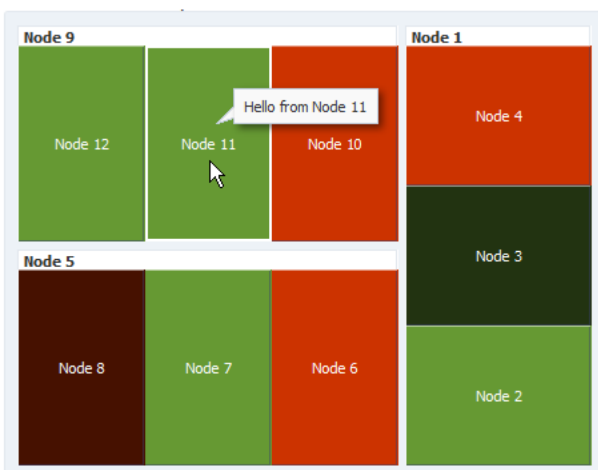


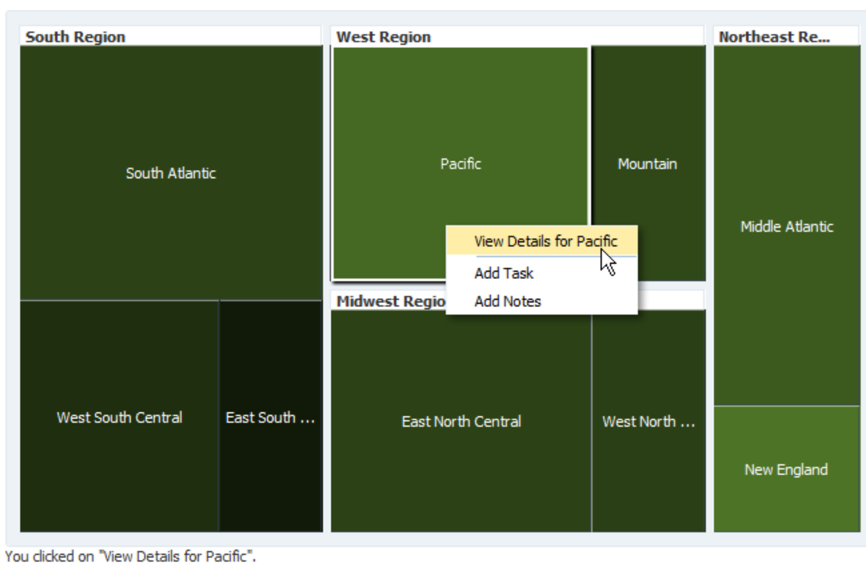
Figure 30–10 shows a similar popup window that is displayed when the user clicks on one of the sunburst nodes.

Figure 30–10 Sunburst Popup on Mouse Click

30.1.2.8 Context Menus

Treemaps and sunbursts support the ability to display context menus to provide additional information about the selected node. Figure 30–11 shows a context menu displayed when the user right-clicks on one of the sunburst nodes.

Figure 30–11 Sunburst Context Menu

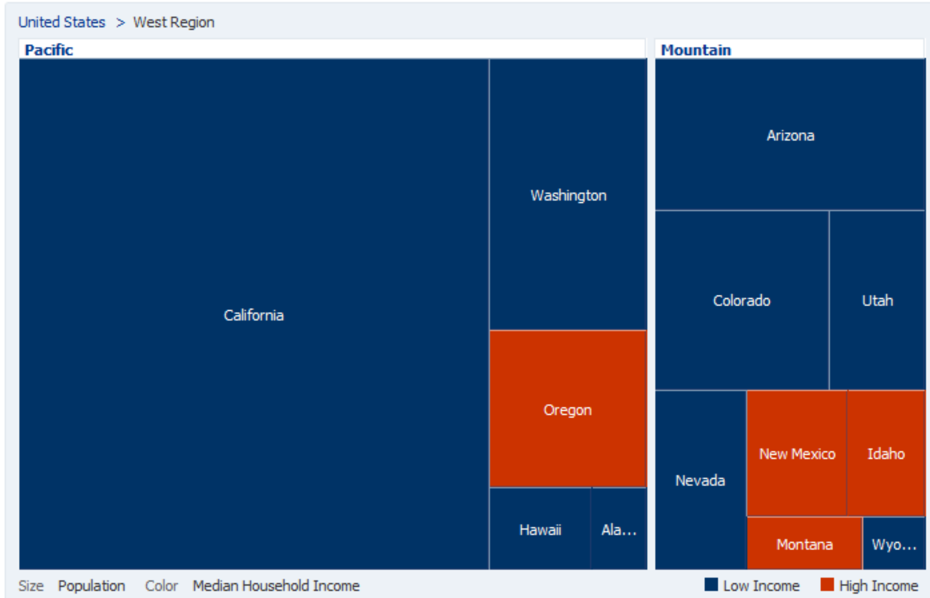


30.1.2.9 Drilling Support

`Treemap` and `sunburst` components support drilling to navigate through the hierarchy and display more detailed information about a node.

Figure 30–12 shows the treemap that is displayed when a user clicks on the West Region header text in Figure 30–5. The user can navigate back up the hierarchy by clicking on the **United States** > **West Region** breadcrumb.

Figure 30–12 Treemap Drilled on a Group Node Header

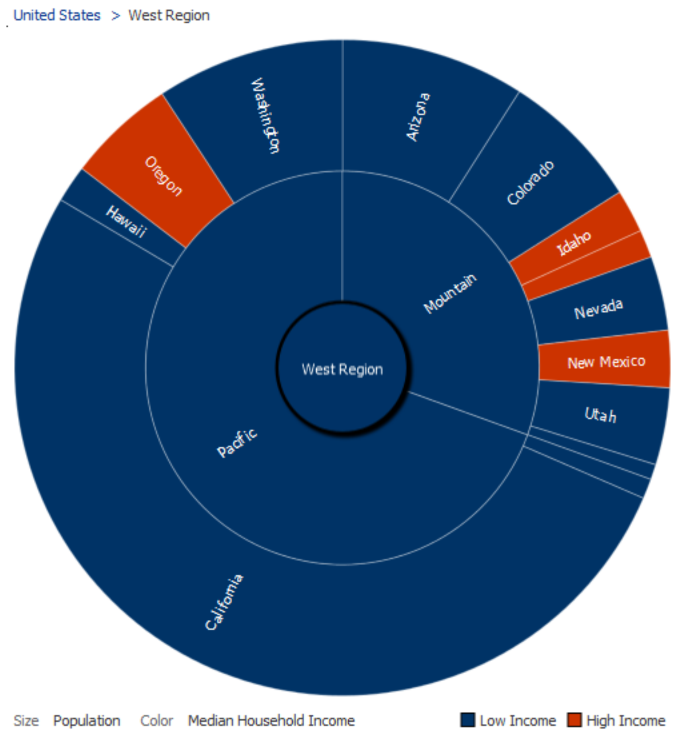


The user can also double-click on a node to set the node as the root of the hierarchy as shown in Figure 30–13.

Figure 30–13 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 30–14. 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.

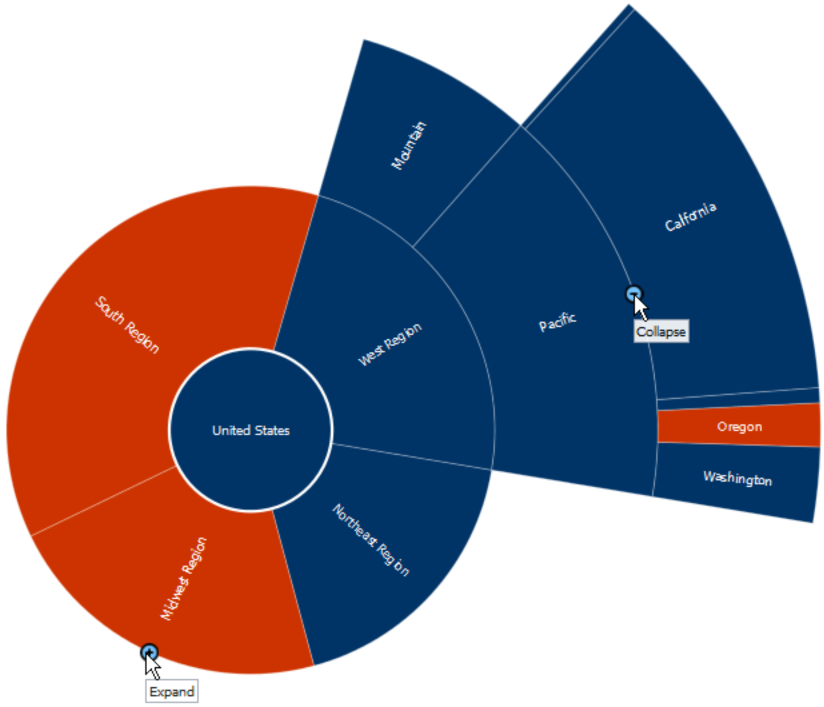
Figure 30–14 Sunburst Drilled on a Node



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 30–15 shows a sunburst configured for asymmetric drilling.

Figure 30–15 Sunburst Configured for Asymmetric Drilling

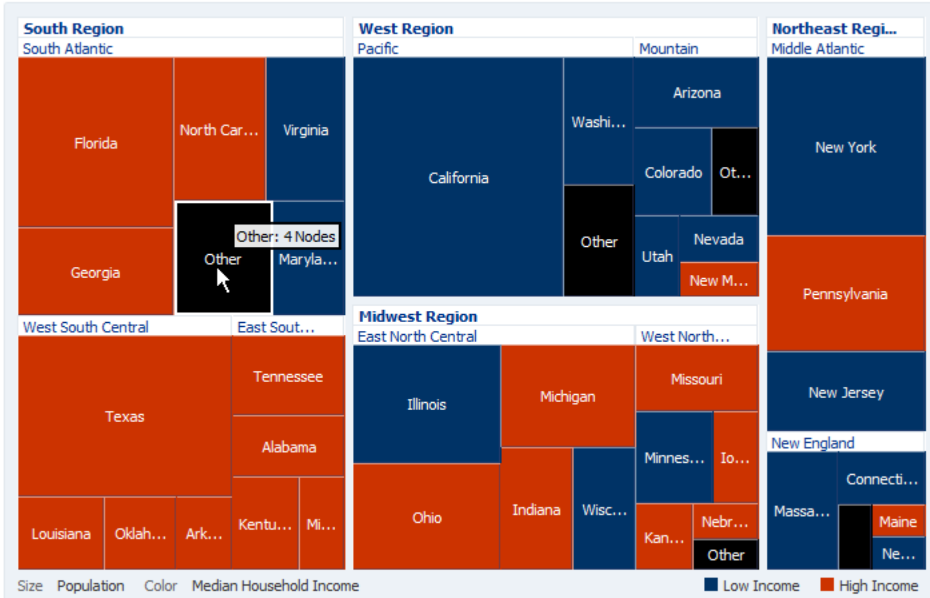


30.1.2.10 Other Node Support

`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 30–16 shows the census treemap displayed in Figure 30–5 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 30–16 Treemap Displaying Other Node



30.1.2.11 Drag and Drop Support

Treemap and sunburst components support drag and drop both as a drop source and a drop target.

Figure 30–17 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 30–17 Treemap Configured as a Drag Source

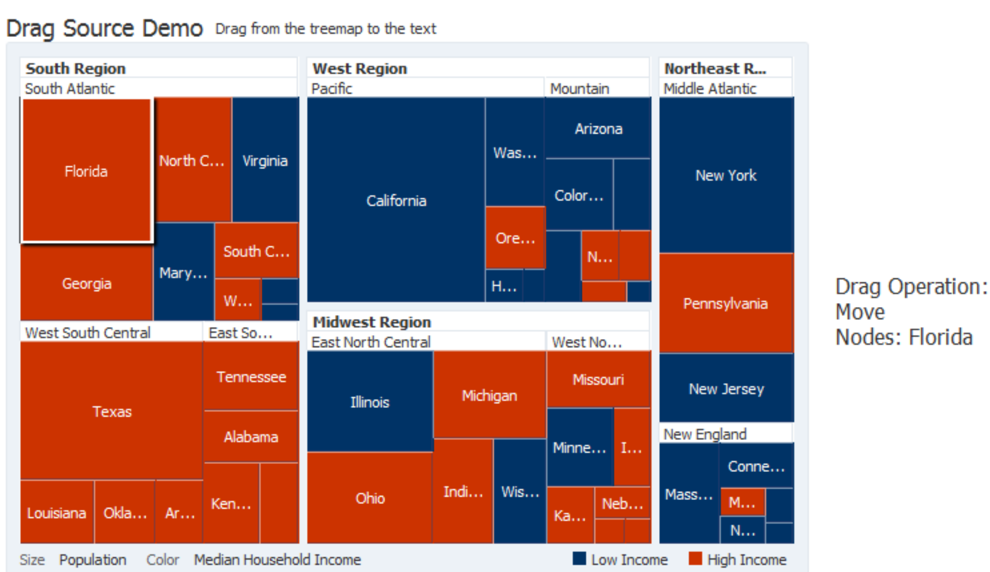
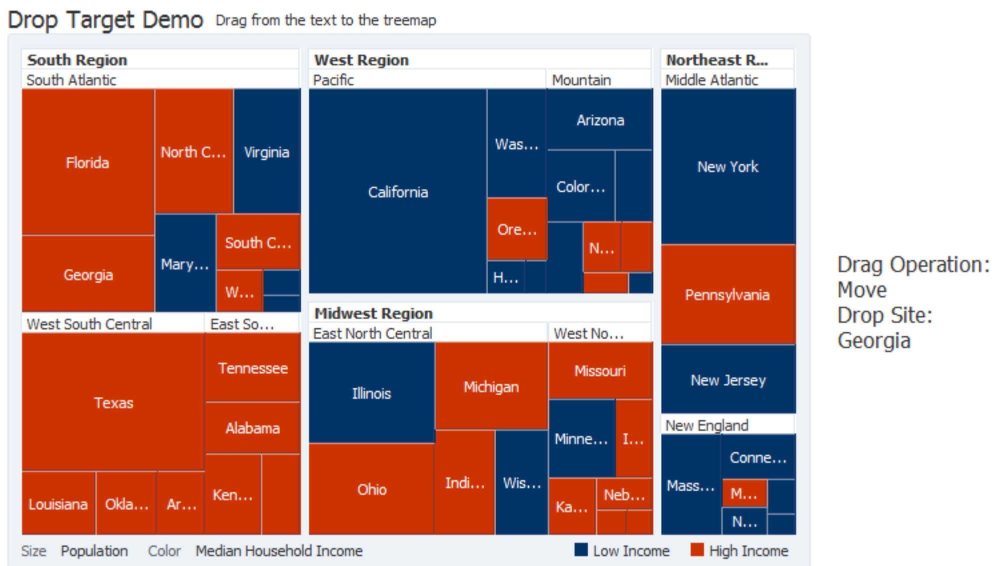


Figure 30–18 shows a treemap configured as a drop target. When the user drags the text on the right to one of the nodes, the text changes to reflect which node received the text.

Figure 30–18 Treemap Configured as a Drop Target

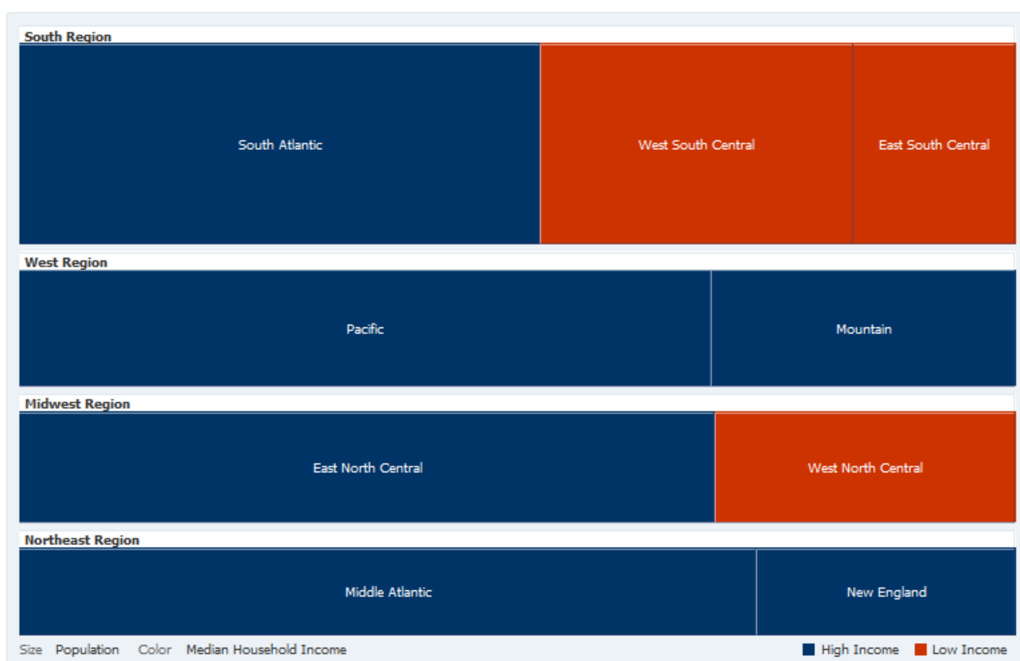


30.1.2.12 Sorting Support

`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 30–19 shows a sorted treemap. The nodes are arranged in decreasing size, making it easy to see which regions have the largest population.

Figure 30–19 Sorted Treemap



Note: Treemaps support sorting in the slice and dice layouts only.

30.1.2.13 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 Section A.4.3, "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

30.1.2.14 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 30–20 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 30–20 Sunburst Displaying Advanced Root Node Content

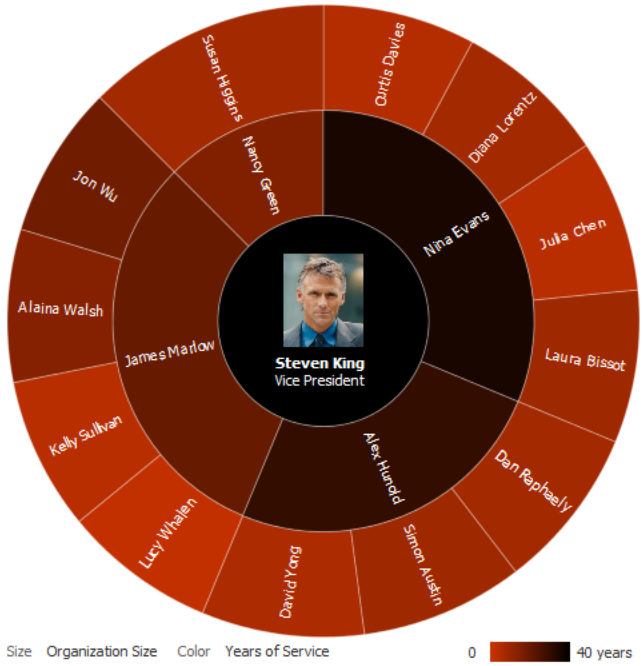
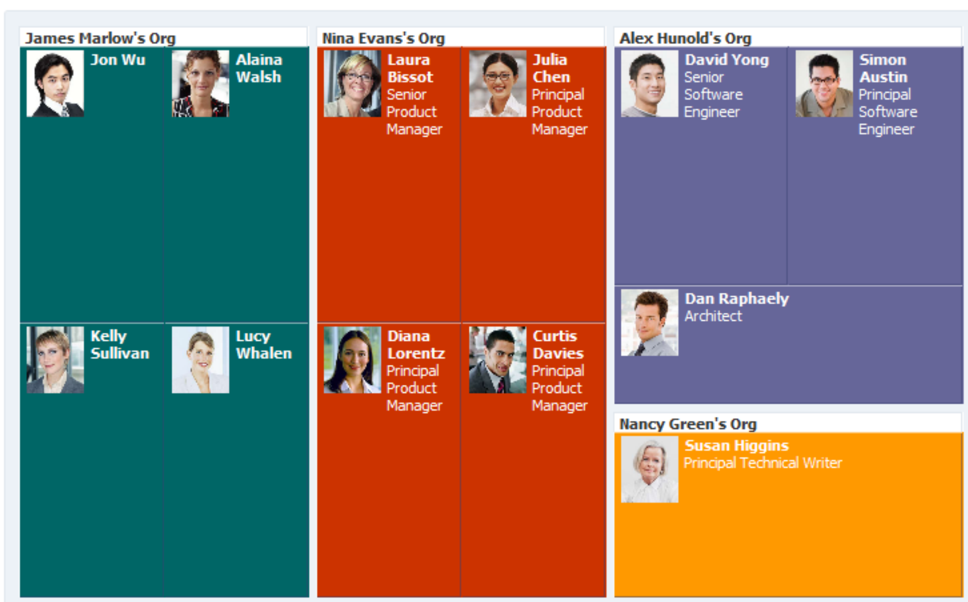


Figure 30–21 shows an example of a treemap using advanced node content.

Figure 30–21 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 Section 30.4.7, "Configuring Treemap and Sunburst Advanced Node Content."

30.1.2.15 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 Chapter 37, "Using Different Output Modes."

30.1.2.16 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 Chapter 38, "Using the Active Data Service with an Asynchronous Backend."

30.1.2.17 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 30–22 shows the **Isolate** icon that appears when the user moves the mouse over the South Atlantic group header.

Figure 30–22 Isolate Icon Displayed on Treemap Group Header

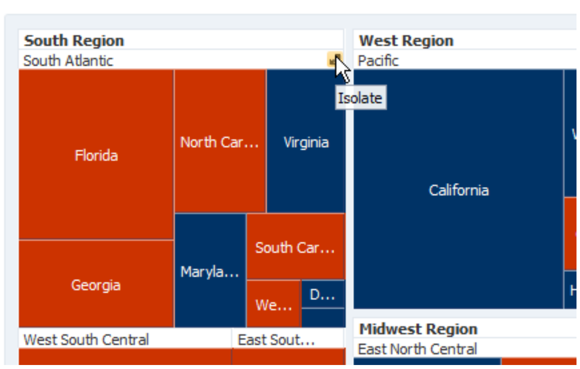
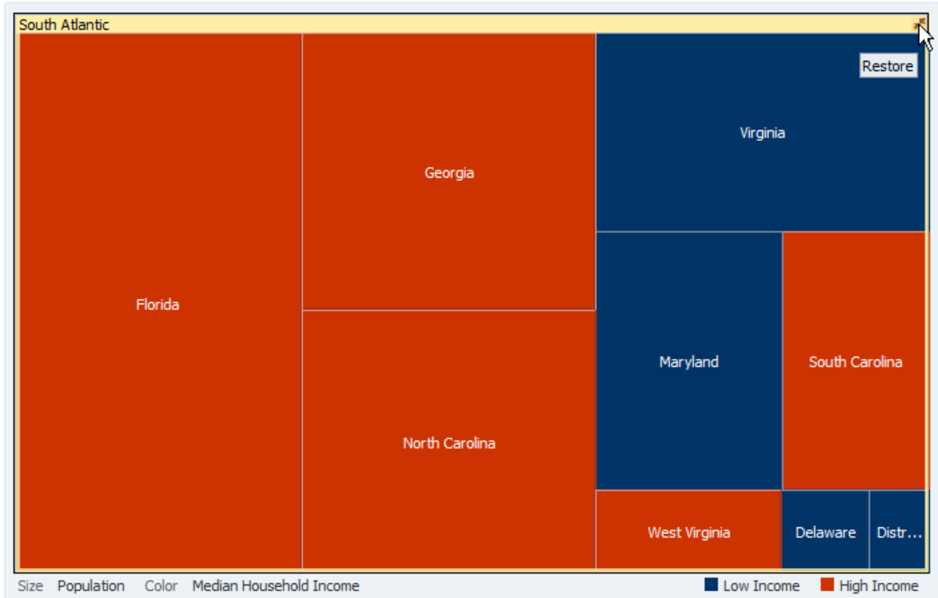


Figure 30–23 shows the treemap that is displayed when the users click the **Isolate** icon for the South Atlantic region in Figure 30–22.

To restore the treemap to the original view, users click the **Restore** icon.

Figure 30–23 Treemap Isolated on a Group

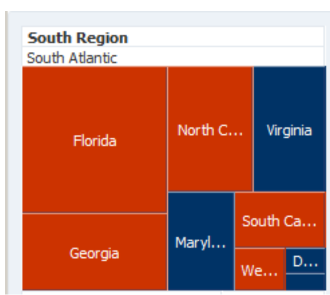


30.1.2.18 Treemap Group Node Header Customization (Treemap Only)

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 30–24 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.

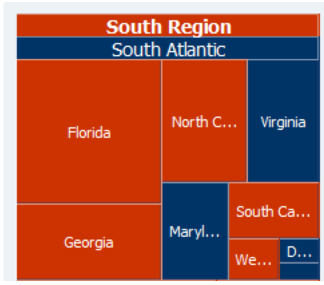
Figure 30–24 Treemap Showing Default Node Header Formatting



You can customize the headers to use the node's color, change the text alignment or customize the font.

Figure 30–25 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 30–25 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.

30.1.3 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 Chapter 8, "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 Chapter 35, "Allowing User Customization on JSF Pages."
- 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 Chapter 31, "Customizing the Appearance Using Styles and Skins."
- Accessibility: You can make your treemap and sunburst components accessible. For more information, see Chapter 33, "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 Appendix D, "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 Section 22.2, "Common Functionality in Data Visualization Components."

30.2 Using the Treemap and Sunburst Components

To use the `treemap` and `sunburst` components, define the data, add the treemap or sunburst to a page and complete the additional configuration in JDeveloper.

30.2.1 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 Section 12.6, "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 30–26 shows a subset of the data used to generate the treemap in Figure 30–1. This is the same data used to generate the sunburst in Figure 30–2.

Figure 30–26 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 Section F.3.1, "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.

30.2.2 Using the Treemap Component

To use the `treemap` component, add the treemap to a page and complete the additional configuration in JDeveloper.

30.2.2.1 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`, `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 to 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.

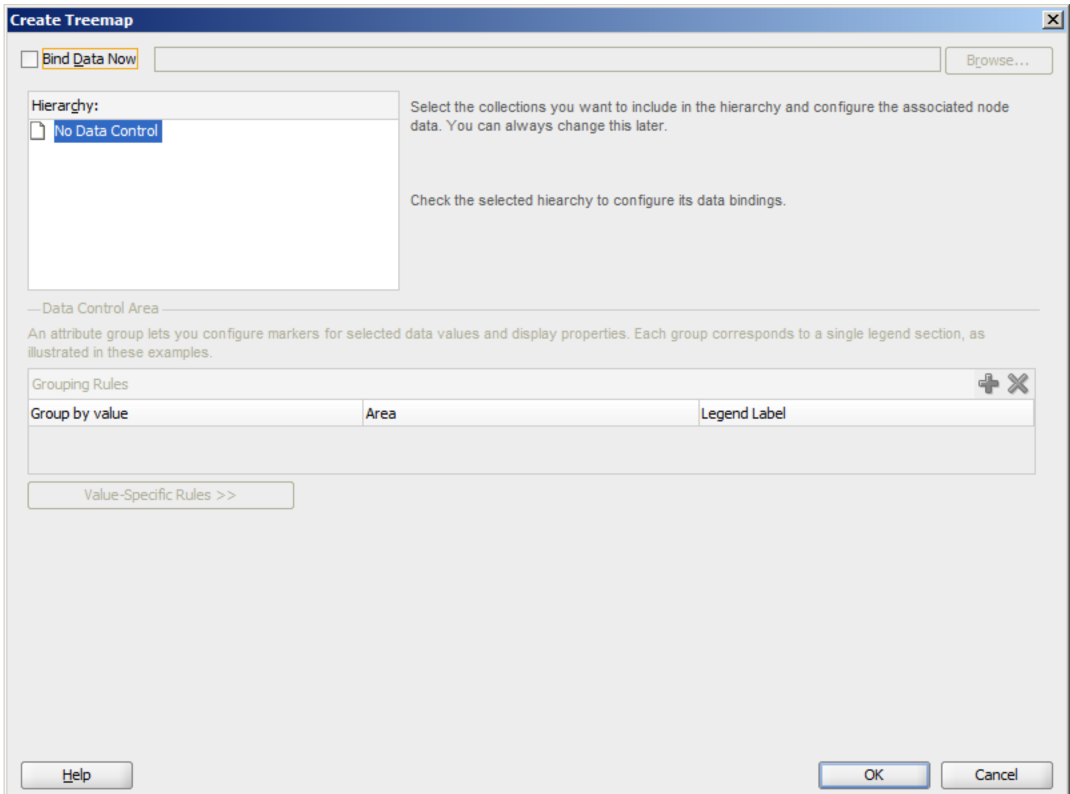
- Supported facets: optional children of the treemap or treemap node. The `treemap` component supports facets for displaying popup components, and the treemap's `node` component supports a content facet for providing additional detail when the treemap node's label is not sufficient.

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 Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

30.2.2.2 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 30–27 shows the Create Treemap dialog.

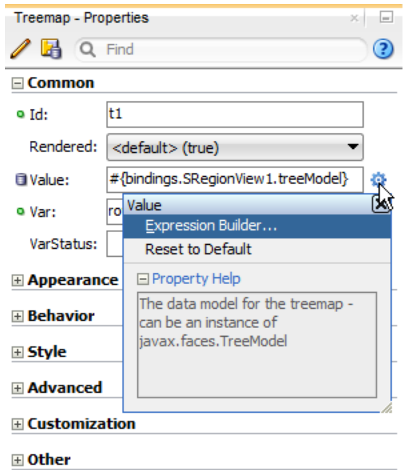
Figure 30–27 Create Treemap Dialog Using UI-First Development



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 30–28 shows the dropdown menu for a treemap `value` attribute.

Figure 30–28 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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

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 values for the following attributes:
   - **Layout**: Use the attribute's dropdown menu to change the default layout from `squarified` to `sliceAndDiceHorizontal` or `sliceAndDiceVertical`.

     See Section 30.1.2.1, "Treemap and Sunburst Layouts" for sample images of treemap layouts.

- **Summary**: Enter a summary of the treemap's purpose and structure for screen reader support.

30.2.2.3 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. Example 30–1 shows the generated code.

Example 30–1 Treemap Sample Code in UI-First Development

```
<dvt:treemap id="t1">
  <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*.

30.2.3 Using the Sunburst Component

To use the `sunburst` component, add the sunburst to a page and complete the additional configuration in JDeveloper.

30.2.3.1 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`, `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 `rotated` which will display the rotated labels inside the node, but you can also set it to `off` to turn off the label display or `on` to display horizontal labels within the nodes.

   Note: Rotated text is not supported on all client technologies. In particular, rotated text is not supported on clients using the Flash image format.

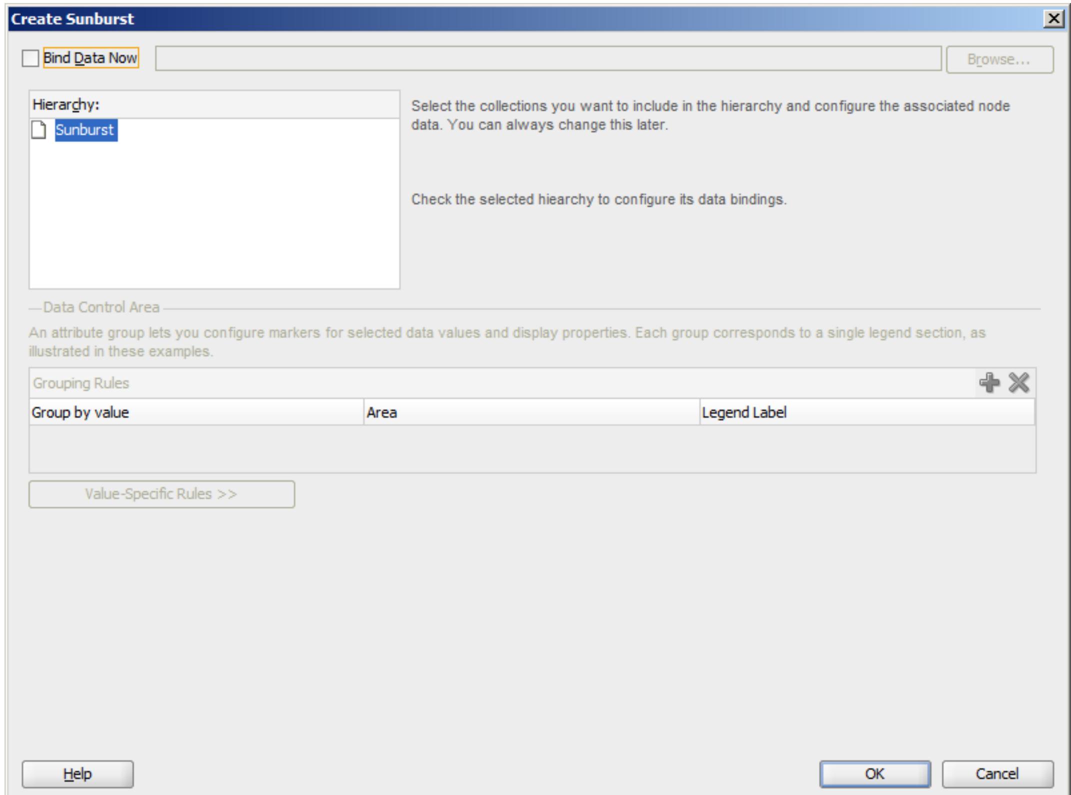
- Attribute group (`dvt:attributeGroup`): optional child of the sunburst node. Add this attribute to generate `fillColor` and `fillPattern` values automatically based on categorical bucketing or continuous classification of the data set.
- Supported facets: optional children of the sunburst or sunburst 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 Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

30.2.3.2 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 30–29 shows the Create Sunburst dialog.

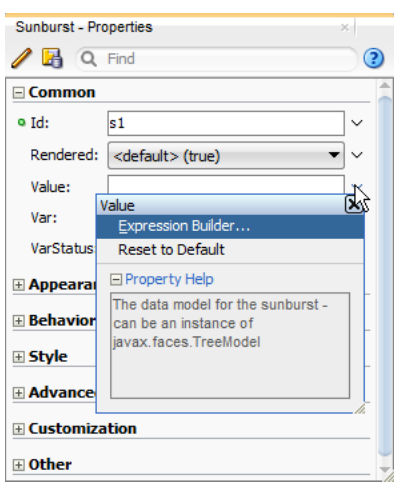
Figure 30–29 Create Sunburst Dialog Using UI-First Development



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 30–30 shows the dropdown menu for a sunburst `value` attribute.

Figure 30–30 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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

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 **Other** section, and set a value for the following attributes:
   - **Summary**: Enter text to describe the sunburst's purpose and structure for screen reader support. Alternatively, choose **Select Text Resource** or **Expression Builder** from the attribute's dropdown menu to select a text resource or EL expression.
   - **Rotation**: Use the dropdown menu to specify whether or not rotation is enabled on the sunburst. By default, this rotation is set to `on`, but you can also set it to `off` to disable rotation. Alternatively, choose **Expression Builder** from the attribute's dropdown menu to create an EL expression that sets the rotation.
   - **StartAngle**: Enter a value for the start angle of the sunburst. By default, the angle is set to `90`, but you can enter any value between 0 and 360. Alternatively, choose **Expression Builder** from the attribute's dropdown menu to create an EL expression that sets the start angle.

30.2.3.3 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. Example 30–2 shows the generated code.

Example 30–2 Sunburst Sample Code in UI-First Development

```
<dvt:sunburst id="s1">
  <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*.

30.3 Adding Data to Treemap and Sunburst Components

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.

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

30.3.1 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page." For help with sunbursts, see Section 30.2.3.2, "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 Section 30.2.1, "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 Section 3.6, "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 Section 3.5.1, "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 Example F–3, "Code Sample to Create a Treemap or Sunburst Tree Node". 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 30–1, "Treemap Displaying United States Population and Median Income by Regions", the fill color is calculated from income data.

     Example 30–3 shows the sample method used by the census data example. To reference this example in the Expression Builder, create the following expression: `#{row.color}`.

Example 30–3 Sample Method to Set Treemap or Sunburst Node Fill 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 Example F–5, "Managed Bean Example to Set Census Data Treemap" 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 Section 30.4.3.1, "How to Configure Treemap and Sunburst Discrete Attribute Groups."

30.3.2 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*.

30.4 Customizing Treemap and Sunburst Display Elements

You can configure treemap and sunburst display elements, including patterns, attribute groups, legends, labels, animation, aggregation of smaller data contributors, skinning, sizing, and ordering of the nodes by size.

30.4.1 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.

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.

Example 30–4 shows the skinning key for a sunburst configured to show the node's text in bold when the user selects it.

Example 30–4 Using a Skinning Key to Change Font Weight When Node is Selected

```
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 Chapter 31, "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 Chapter 9, "Organizing Content on Web Pages."

30.4.2 Configuring Pattern Display

You can configure the treemap or sunburst node to display patterns. 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 Section 30.4.3.1, "How to Configure Treemap and Sunburst Discrete Attribute Groups."

30.4.3 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.

30.4.3.1 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 30–5, create an expression similar to the following expression for the **Value** field:

```
#{row.income > 50000}
```

For help with creating EL expressions, see Section 3.5.1, "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:

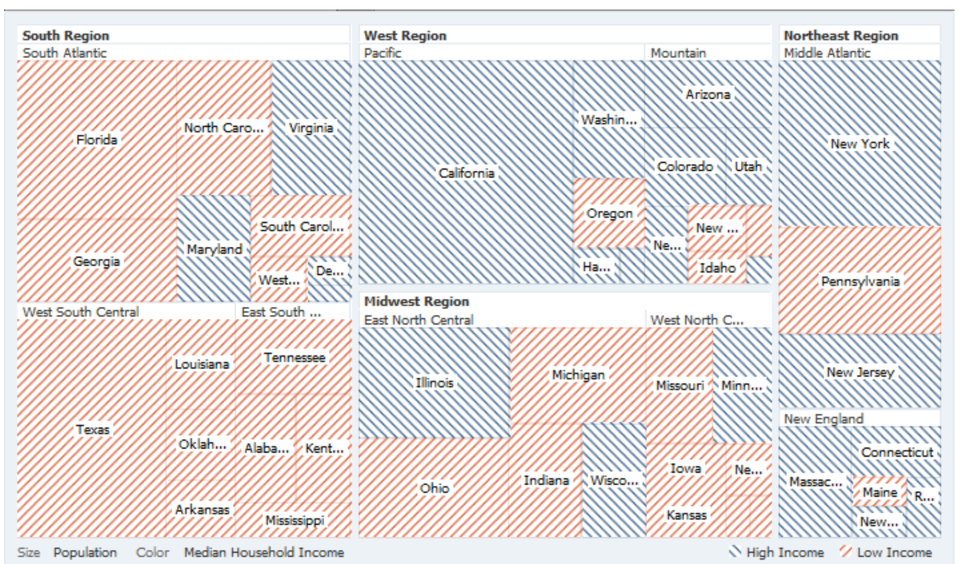
   ```
   #{row.income > 50000 ? 'High Income' : 'Low Income'}
   ```

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 30–31 shows the treemap that displays if you accept the default colors and patterns in the census data example.

Figure 30–31 Treemap Discrete Attribute Group with Default Colors and Patterns



8. Optionally, to change the attribute group's default colors, do the following:
   1. 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.

   2. Right-click the **dvt:attributeMatchRule** element and choose **Go to Properties**.

3. 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.

4. In the Structure window, right-click the **dvt:attributeMatchRule** element and choose **Insert Inside Match Rule > Attribute**.

5. 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`.

6. Repeat step 1 through step 5 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:

   1. In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups > Attribute Match Rule**.

   2. Right-click the **dvt:attributeMatchRule** element and choose **Go to Properties**.

   3. 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.

   4. In the Structure window, right-click the **dvt:attributeMatchRule** element and choose **Insert Inside Attribute Match Rule > Attribute**.

   5. 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 Section 30.4.2, "Configuring Pattern Display."

   6. Repeat step 1 through step 5 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`.

Example 30–5 shows the code on the JSF page if you configure a discrete attribute group for the treemap shown in Figure 30–5, "Treemap Displaying Discrete Attribute Groups".

Example 30–5 Sample Code on JSF Page for Discrete Attribute Group

```
<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>
```

30.4.3.2 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 Section 3.5.1, "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:
   1. In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups** > **Attribute**.
   2. 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 black, which is the color used in the attribute group in Figure 30–1, enter the following in the **value** field: `#000000`.
   3. In the Structure window, right-click the **dvt:attributeGroups** element and choose **Insert Inside Attribute Groups** > **Attribute**.
   4. 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 30–1, enter the following in the **value** field: `#00AA00`.

Example 30–8 shows the code on the JSF page if you configure the continuous attribute group shown in Figure 30–1.

Example 30–6 Sample Code on JSF Page for Continuous Attribute Group

```
<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="#000000"/>
        <f:attribute name="color2" value="#00AA00"/>
      </dvt:attributeGroups>
      <f:facet name="content"/>
    </dvt:treemapNode>
</dvt:treemap>
```

30.4.3.3 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 Section 30.4.5.2, "What You May Need to Know About Configuring the Treemap and Sunburst Other Node."

30.4.4 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 Section 3.5.1, "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:
   1. From the **LegendSource** property's dropdown menu, choose **Edit**.
   2. In the Edit Property: LegendSource dialog, expand each component and locate the `attributeGroups` component.
   3. Select the `attributeGroups` component and click **OK**.

30.4.5 Configuring the Treemap and Sunburst Other Node

Use the **Other** node to aggregate smaller data sets visually into one larger set for easier comparison.

30.4.5.1 How to Configure the Treemap and Sunburst Other Node

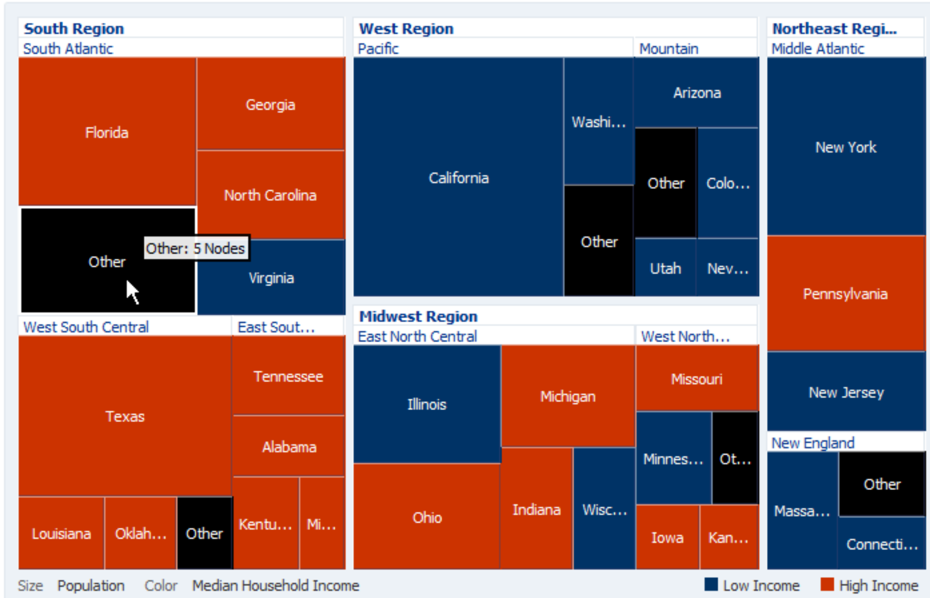
Configure the treemap **Other** node by setting values for the following attributes:

- **OtherThreshold**: Specify the percentage of the parent 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 which are less than 10% of their parent to be aggregated into the **Other** node. In Figure 30–16, 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 `.1` or 10%, the **Maryland** node is added to the aggregation. Figure 30–32 shows the same treemap with the **otherThreshold** attribute set to `.1`.

Figure 30–32 Treemap Showing Other Node With otherThreshold Set to 10 Percent



- **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 30–32 to reflect the higher mean income when the **Maryland** node is included in the **Other** node.

  Example 30–7 shows the sample method to specify the `otherColor` value based on the mean income in the census data example.

Example 30–7 Sample Method to Set Treemap otherColor Value

```
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.

  Example 30–8 shows the sample code for a method that sets the pattern fill to `smallDiamond` on the **Other** node.

Example 30–8 Sample Method to Set Treemap or Sunburst otherPattern Value

```
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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 method in Example 30–7 to a managed bean.

  If you need help with managed beans, see Section 3.6, "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 method in Example 30–8 to a managed bean.

To add the otherColor 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.
- **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}`.

30.4.5.2 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.
- `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.

30.4.6 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.

30.4.7 Configuring Treemap and Sunburst Advanced Node Content

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.

30.4.7.1 How to Add Advanced Node Content to a Treemap

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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a treemap to your page. For more information, see Section 30.2.2.2, "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 Section 9.13.1, "How to Use the panelGroupLayout Component."

   For help with configuring images and output text, see Chapter 18, "Using Output Components."

30.4.7.2 How to Add Advanced Root Node Content to a Sunburst:

Configure advanced node content on a treemap 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 Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a sunburst to your page. For more information, see Section 30.2.3.2, "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 Section 9.13.1, "How to Use the panelGroupLayout Component."

   For help with configuring images and output text, see Chapter 18, "Using Output Components."

30.4.7.3 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.

30.4.8 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a treemap or sunburst to your page. For more information, see Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 30–1 shows the list of supported animation effects.

Table 30–1 Treemap and Sunburst Animation Effects

Animation Effect	AnimationOnDisplay	AnimationOnDataChange
`none`	x	x
`activeData`		x
`alphaFade`	x	x
`auto`		x
`cubeToLeft`		x (treemap only)
`cubeToRight`		x (treemap only)
`fan`	x (sunburst only)	
`flipLeft`		x (sunburst only)
`flipRight`		x (sunburst only)
`slideToLeft`		x
`slideToRight`		x
`transitionToLeft`		x
`transitionToRight`		x
`zoom`	x	x

30.4.9 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 Section 30.4.4, "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 Section 30.4.10, "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.

30.4.9.1 How to Configure Treemap Leaf Node Labels

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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a treemap to your page. For more information, see Section 30.2.2.2, "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 more, 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.

30.4.9.2 How to Configure Sunburst Node Labels

Configure sunburst node labels by setting 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a sunburst to your page. For more information, see Section 30.2.3.2, "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 rotated which displays rotated labels within the nodes if the client's environment supports rotated text. You can also set this to `off` to turn off the label display or to `on` which will display horizontal 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.

30.4.10 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.

30.4.10.1 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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

You will need to complete these tasks:

- Add a treemap to your page. For more information, see Section 30.2.2.2, "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 Section 30.3, "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`.

30.4.10.2 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.

30.4.10.3 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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a treemap to your page. For more information, see Section 30.2.2.2, "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.

30.5 Adding Interactive Features to Treemaps and Sunbursts

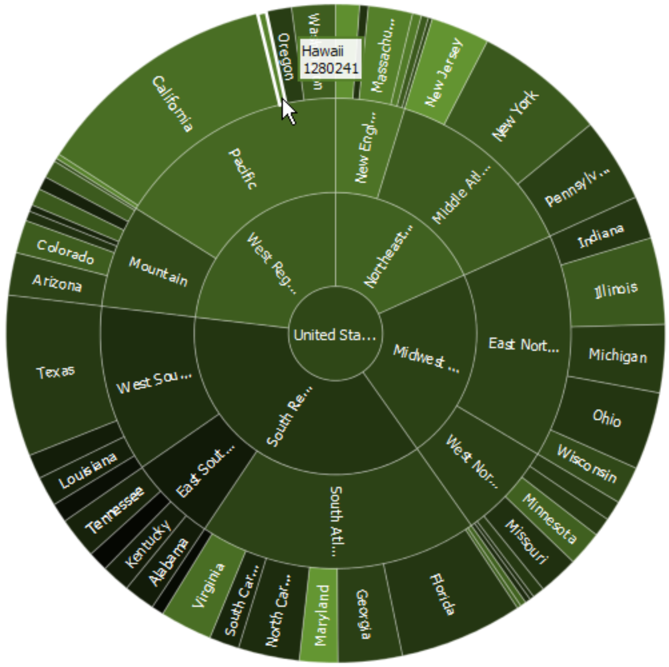
You can add interactive features to treemaps and sunbursts, including tooltips, popups, selection support, context menus, and drilling. Treemaps also provide support for isolation of group nodes.

30.5.1 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 30–33 shows a sunburst displaying the name and size of one of the sunburst nodes.

Figure 30–33 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}
```

30.5.2 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 Section 30.5.4, "Configuring Treemap and Sunburst Context Menus" to see how to display a context menu using the `af:popup` component.

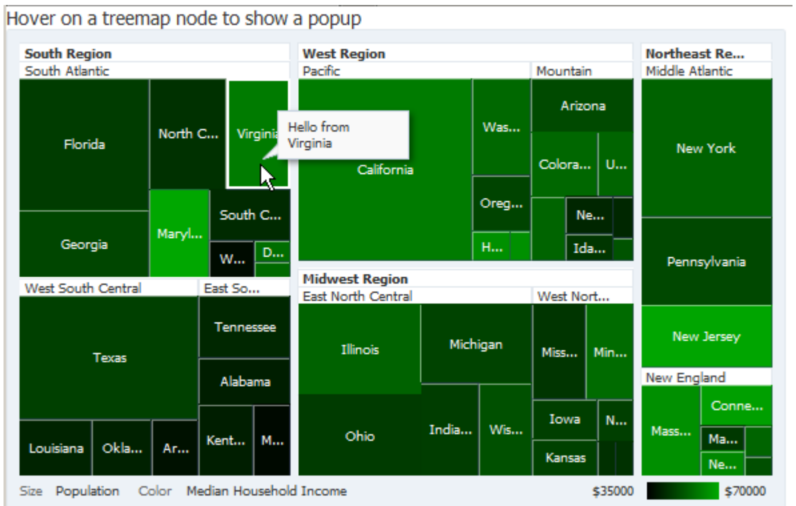
30.5.2.1 How to Add Popups to Treemap and Sunburst Components

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 30–34 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 30–34 Treemap Showing Popup on Mouse Hover



Example 30–9 shows the code on the page to declare the popup.

Example 30–9 Sample Code for Treemap Popup on Mouse Hover

```
<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"
               inlineStyle="width:450px; height:350px;"
               summary="Treemap Popup"
               displayLevelsChildren="3">
       <dvt:treemapNode id="tn1" value="#{row.size}" fillColor="#{row.color}
               label="#{row.text}">
          <af:showPopupBehavior popupId="::noteWindowPopup"
                              triggerType="mouseHover"/>
       </dvt:treemapNode>
    </dvt:treemap>
  </af:panelGroupLayout>
  <af:popup childCreation="deferred" autoCancel="disabled"
          id="noteWindowPopup" launcherVar="source" eventContext="launcher"
          clientComponent="true" contentDelivery="lazyUncached">
    <af:setListener 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 30–34 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 Example 30–9.

  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 Example 30–10 to the treemap bean:

Example 30–10 Code Fragment to Add noteWindowMessage Variable to Treemap Managed 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 Section 3.6, "Creating and Using Managed Beans." For additional details about using popup windows to display dialogs, menus, and windows, see Chapter 15, "Using Popup Dialogs, Menus, and Windows."

- Create any additional components needed to display the selection.

  For example, the page in Figure 30–34 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 Section 18.2, "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`.

30.5.2.2 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 Section 30.4.5.2, "What You May Need to Know About Configuring the Treemap and Sunburst Other Node."

30.5.3 Configuring Treemap and Sunburst Selection Support

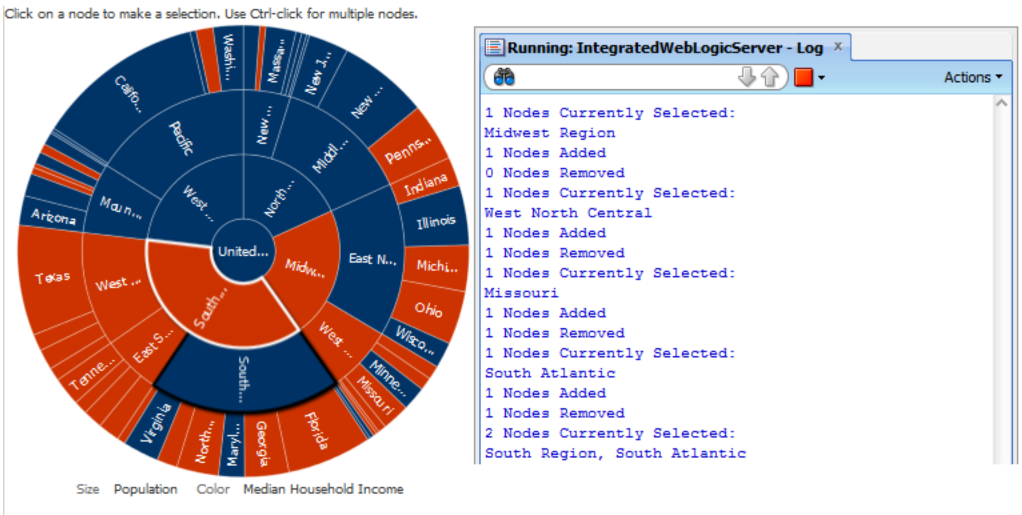
The `treemap` and `sunburst` components support single or multiple node selection. If the component allows multiple selections, users can select multiple nodes using a Control+click operation.

30.5.3.1 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 30–35 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 30–35 Sunburst Illustrating Custom Selection Listener



Example 30–11 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.

Example 30–11 Code Sample for Sunburst selectionListener Method

```
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. Example 30–12 shows the portion of the page used to set up the sunburst. The `selectionListener` attribute is highlighted in bold font.

Example 30–12 Sunburst Sample Page Declaring Selection Listener

```
<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"
                  colorLabel="Income"
                  sizeLabel="Population" displayLevelsChildren="3"
                  selectionListener="#{sunburst.selectionListener}">
       <dvt:sunburstNode id="sn1" value="#{row.size}" fillColor="#{row.color}"
                  label="#{row.text}"
                  shortDesc="#{row.text}&lt;br/>#{row.size}"/>
    </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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 Section 3.6, "Creating and Using Managed Beans."
- Create any additional components needed to display the selection.

  For example, the page in Figure 30–35 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 Section 18.2, "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}`.

30.5.3.2 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 Section 12.6.3, "What Happens at Runtime: Tree Component Events."

For additional information about event handling in JDeveloper, see Chapter 6, "Handling Events."

30.5.4 Configuring Treemap and Sunburst Context Menus

You can configure both treemaps and sunbursts to display context menus when a user right-clicks a node.

30.5.4.1 How to Configure Treemap and Sunburst Context Menus

Define treemap and sunburst context menus using these context menu facets:

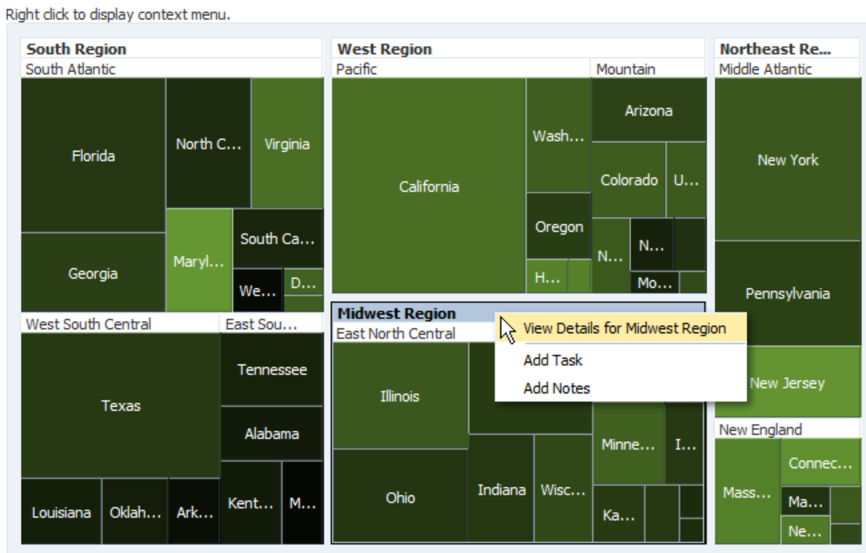
- `bodyContextMenu`: Specifies a context menu that is displayed on non-selectable elements in the treemap or sunburst component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the treemap or sunburst component.
- `multiSelectContextMenu`: Specifies a content 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 Chapter 15, "Using Popup Dialogs, Menus, and Windows."

Figure 30–36 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.

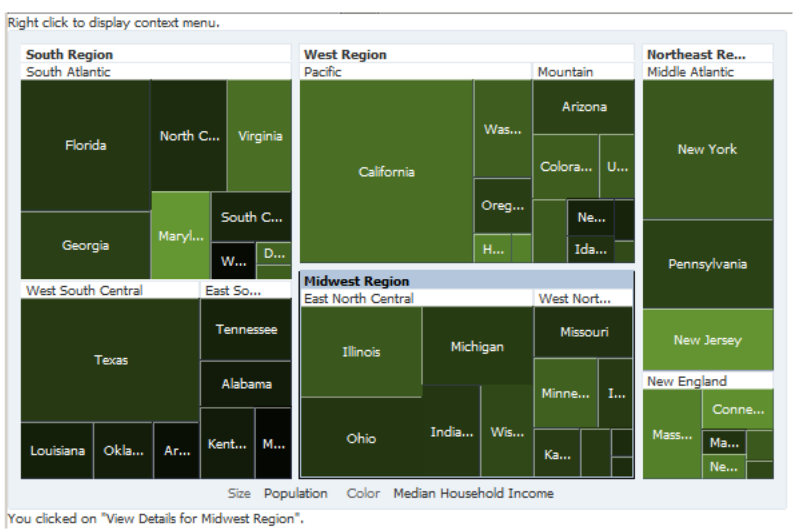
Figure 30–36 Treemap Context Menu



If the user selects **View Details for Midwest Region**, the application can provide additional information about the **Midwest Region** node.

Figure 30–37 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 30–37 Context Menu Sample Output After Click



Example 30–13 shows the sample code used to configure the example treemap and the context menu.

Example 30–13 Code Sample for Treemap 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}"
```

```
               binding="#{treemapContextMenu.treemap}">
    <dvt:treemapNode id="tn1" value="#{row.size}" fillColor="#{row.color}"
                     label="#{row.text}"/>
    <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="cmi1"
                       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: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 Section 30.3, "Adding Data to Treemap and Sunburst Components." Example 30–14 shows the code for the `ContextMenuSample` class.

Example 30–14 ContextMenuSample Class Code

```
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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 in Example 30–14, "ContextMenuSample Class Code" into a backing bean named `treemapContextMenu`. If you need help with managed beans, see Section 3.6, "Creating and Using Managed Beans."
- Create any additional components needed to support the context menu.

  For example, the page in Figure 30–36 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 in Example 30–14 for the details needed to configure the additional components. For additional information about `af:outputText` and `af:outputFormatted` components, see Section 18.2, "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:
    1. In the Structure window, right-click the **dvt:treemap** or **dvt:sunburst** node and choose **Go to Properties**.
    2. 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 Section 30.3.1, "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 Example 30–13. For information about the `af:group` component, see Section 9.13, "Grouping Related Items."

13. To configure additional context menu facets, repeat Step 2 through Step 12.

30.5.4.2 What You May Need to Know About Configuring Treemap and Sunburst Context Menus

Context menus are supported in HTML5 and Flash image formats only. When context menus are rendered in Flash, the context menus use the Flash Player's context menu. This imposes several limitations on the context menus. For additional detail about Flash and context menus, see Section 22.2.4, "Context Menus for Graphs, Gauges, Treemaps, and Sunbursts."

30.5.5 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.

30.5.5.1 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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "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 Chapter 6, "Handling Events." If you need help with beans, see Section 3.6, "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:
   1. In the Structure window, right-click the **dvt:treemap** node and choose **Go to Properties**.
   2. In the Properties window, expand the **Behavior** section.
   3. From the **DrillListener** attribute's dropdown menu, choose **Edit**.
   4. In the Edit Property dialog, use the search box to locate the treemap's managed bean.
   5. Expand the managed bean node and select the method that contains the drill listener.
   6. 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.

30.5.5.2 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 Section 30.5.7, "Configuring Isolation

Support (Treemap Only)."

30.5.6 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 Chapter 36, "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 Section 30.2.2.1, "Configuring Treemaps." For information about configuring sunburst attributes and child tags, see Section 30.2.3.1, "Configuring Sunbursts."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "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 Section 30.2.2.2, "How to Add a Treemap to a Page" or Section 30.2.3.2, "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 Section 30.3, "Adding Data to Treemap and Sunburst Components."
- Create any additional components needed to support the drag and drop.

  For example, the page in Figure 30–17 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.

  Example 30–15 shows the sample code for the completed page. For additional information about `af:outputText` and `af:outputFormatted` components, see Section 18.2, "Displaying Output Text and Formatted Output Text."

Example 30–15 Sample Code for Treemap Drag Source Example

```
<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="MOVE" 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>
```

Example 30–16 shows the sample code for the page in Figure 30–18. In this example, the treemap is configured as the drop target.

Example 30–16 Sample Code for Treemap Drop Target Example

```
<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.

   Example 30–17 shows the sample drop listener and supporting methods for the treemap displayed in Figure 30–18.

Example 30–17 Sample Drop Listener for a Treemap

```
// 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:

1. 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.

2. 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 30–17, "Treemap Configured as a Drag Source" uses `MOVE` as the default action.

11. To make another component the drop target for drags from the treemap or sunburst, do the following:

    1. 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 30–18, "Treemap Configured as a Drop Target" contains an `af:outputFormatted` component that displays the results of the drop.

    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 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.

       Example 30–18 shows the sample drop listener for the treemap displayed in Figure 30–17. This example uses the same imports and helper methods used in Example 30–17, and they are not included here.

Example 30–18 Sample Drop Listener for a Component Using a Treemap as a Drag Source

```
// 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();
}
```

3. Click OK to enter the Insert Data Flavor dialog.
4. 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.
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. 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}
```

30.5.7 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.

30.5.7.1 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 Section 30.2.2.1, "Configuring Treemaps."

You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 30.1.3, "Additional Functionality for Treemap and Sunburst Components."

Add a treemap to your page. For more information, see Section 30.2.2.2, "How to Add a Treemap to a Page."

Add treemap node headers to your treemap. For more information, see Section 30.4.10.1, "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.

30.5.7.2 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 Section 30.5.5.1, "How to Configure Treemap and Sunburst Drilling Support."

Part VI

Completing Your View

Part VI contains the following chapters:

31

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 ADF Skin Editor, a tool 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:

- Section 31.1, "About Customizing the Appearance Using Styles and Skins"
- Section 31.2, "Changing the Style Properties of a Component"
- Section 31.3, "Enabling End Users to Change an Application's ADF Skin"

31.1 About Customizing the Appearance Using Styles and Skins

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 Section 31.2, "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 31–1 and Figure 31–2 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 31–1 shows a page from the File Explorer application with the `simple` ADF skin applied.

Figure 31–1 Index Page Using the Simple Skin

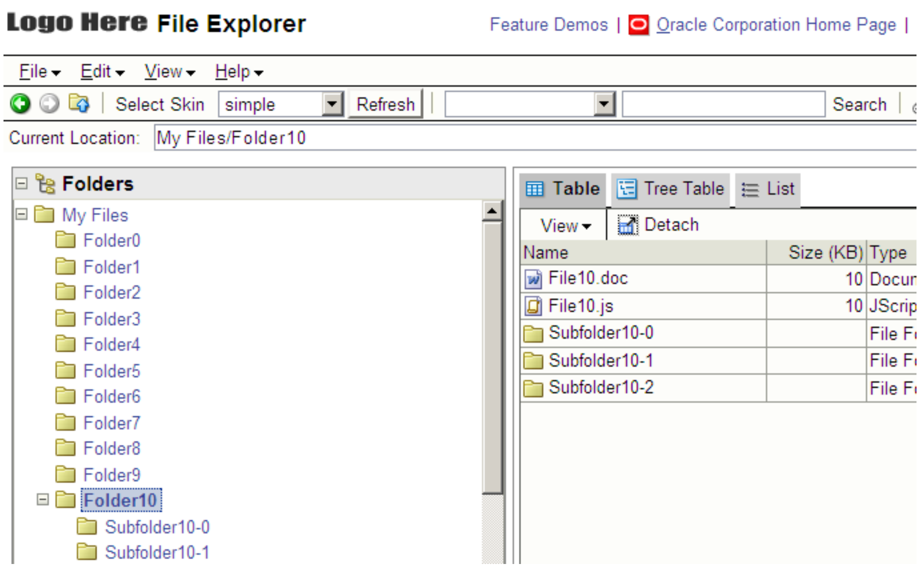
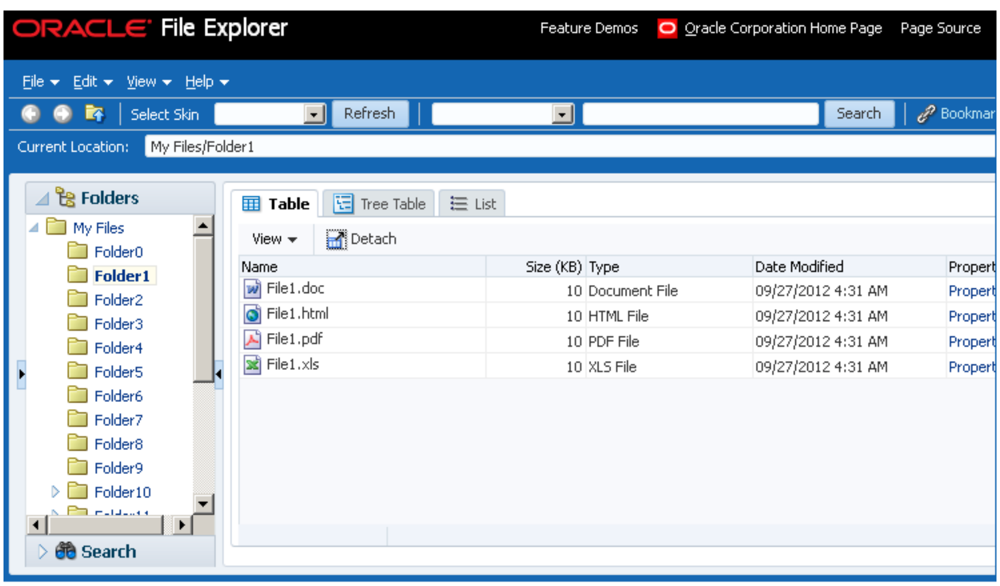


Figure 31–2 shows the same page from the same application with the `skyros` ADF skin applied.

Figure 31–2 Index Page Using the Skyros Skin



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 Section 2.1, "About the ADF Faces Demonstration 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/`

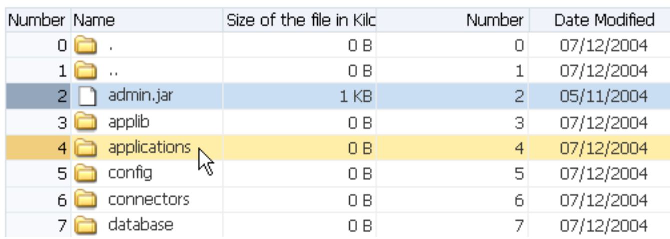
It is also beyond the scope of this guide to describe how to create, modify, or apply ADF skins to your application. For more information, see *Creating ADF Skins with Oracle ADF Skin Editor*.

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 Section 31.3, "Enabling End Users to Change an Application's ADF Skin."

31.1.1 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 31–3.

Figure 31–3 ADF Skin Properties in an ADF Table Component



Number	Name	Size of the file in Kilc	Number	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

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 31–1 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 31–1 Using an ADF Skin Property to Specify Length of Spin Animation

```
af|carousel {
  -tr-spin-animation-duration: 500;
}
```

31.1.2 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 Section 3.5.2, "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 Chapter 32, "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 Chapter 33, "Developing Accessible ADF Faces Pages."

31.2 Changing the Style Properties of a Component

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.

Given a specific selector, you can get style properties for a custom component by creating a class for a renderer. For more information, see Section 34.4.7, "How to Create a Class for a Renderer."

WARNING: 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 Section 9.2.1, "Geometry Management and Component Stretching."

31.2.1 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 Section 9.4, "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 Section 31.2, "Changing the Style Properties of a Component."

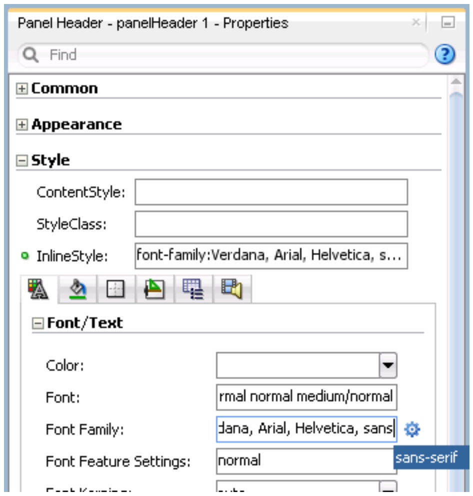
You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 31.1.2, "Additional Functionality for Customizing the Appearance."

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 31–4.

Figure 31–4 Setting an inlineStyle Attribute



JDeveloper adds the corresponding code for the component to the JSF page. Example 31–2 shows the source for an `af:outputText` component with an `inlineStyle` attribute.

Example 31–2 InlineStyle in the Page Source

```
<af:outputText value="outputText1" id="ot1"
       inlineStyle="color:Red; text-decoration:overline;"/>
```

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 31–3.

Example 31–3 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"/>
```

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 31–4.

Example 31–4 Using the contentStyle Property

```
<af:inputText value="outputText1"
          contentStyle="color:Red;" id="it1"/>
```

31.2.2 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 Section 31.2, "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 Section 31.1.2, "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 31–5 shows an example of a style class being used in the page source.

Example 31–5 Page Source for Using a Style Class

```
<af:outputText value="Text with a style class"
               styleClass="overdue"  id="ot4"/>
```

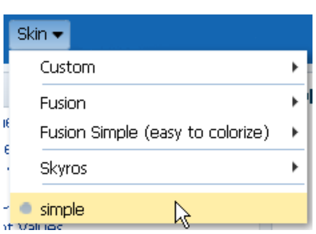
3. You can also use EL expressions for the `styleClass` attribute to conditionally set style attributes. For example, if you want the date to be displayed in red when an action has not yet been completed, you could use code similar to that in Example 31–3.

31.3 Enabling End Users to Change an Application's ADF Skin

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 31–5 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 31–5 Changing an Application's ADF Skin



31.3.1 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 Section 31.3, "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 Section 31.1.2, "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 31–5.

   Example 31–6 shows how you configure `af:commandMenuItem` components that allow end users to choose available ADF skins at runtime, as shown in Figure 31–5. 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.

Example 31–6 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:commandMenuItem id="skin6" text="skin1" type="radio"
    actionListener="#{skins.skinMenuAction}"
    selected="#{skins.skinFamily=='skin1'}"/>
</af:menu>
```

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 31–7 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 31–7 also invokes a method to reload the page after it sets the value of `skinFamily`.

Example 31–7 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();
}
```

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>
```

31.3.2 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 31–6, 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.

32

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:

32.1 About Internationalizing and Localizing ADF Faces Pages

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 Section 32.3.4, "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 Section 32.2, "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.

32.1.1 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 Example 32–1.

Example 32–1 Resource Bundle Element in JSF Configuration File

```
<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 Example 32–2.

Example 32–2 Component Text Referencing Resource Bundle

```
<af:panelBox text="#{res['myDemo.pageTitle']}"
    ....
    id="pb1">
```

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 32–3 shows the keys for the myDemo page.

Example 32–3 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
myDemo.menubar.closedLink=Closed Requests
myDemo.menubar.allRequests=All Requests
myDemo.menubar.newLink=Create New Purchase Request
myDemo.selectAnd=Select and
myDemo.buttonbar.view=View
myDemo.buttonbar.edit=Edit
```

Note that text in the banner image and data retrieved from the database are not translated.

Example 32–4 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.

Example 32–4 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
myDemo.menubar.closedLink=Ticket Risolti
myDemo.menubar.allRequests=Tutti i Ticket
myDemo.menubar.newLink=Creare Nuovo Ticket
myDemo.selectAnd=Seleziona e
myDemo.buttonbar.view=Vedere Dettagli
myDemo.buttonbar.edit=Aggiorna
```

32.1.2 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 Section 3.5.2, "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 Chapter 31, "Customizing the Appearance Using Styles and Skins."
- **Accessibility**: You can make the pages in your application accessible. For more information, see Chapter 33, "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*.

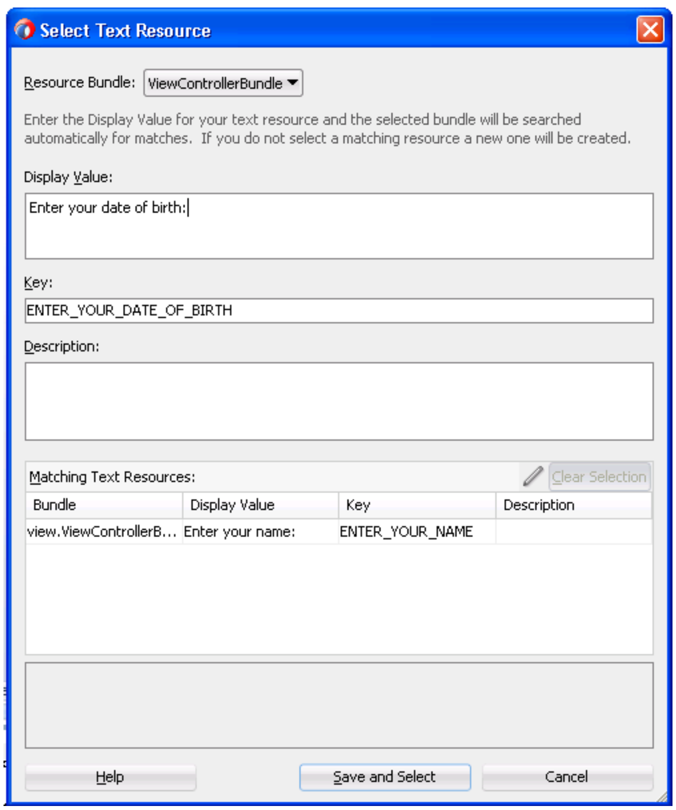
32.2 Using Automatic Resource Bundle Integration in JDeveloper

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 Section 32.2.1, "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 32–1. 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 Section 32.2.3, "How to Create an Entry in a JDeveloper-Generated Resource Bundle."

Figure 32–1 Select Text Resource Dialog



32.2.1 How to Set Resource Bundle Options

You can set resource bundle options for your project in the Project Properties dialog.

Before you begin:

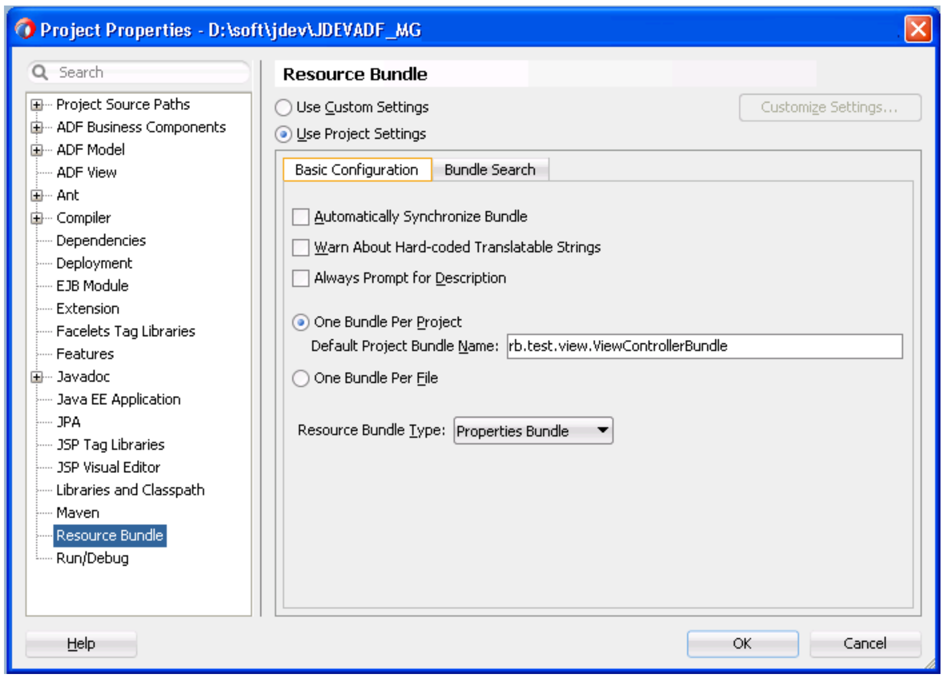
It may help to understand how JDeveloper manages resource bundles. For more information, see Section 32.2, "Using Automatic Resource Bundle Integration in JDeveloper."

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Section 32.1.2, "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 32–2.

Figure 32–2 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**.

32.2.2 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 32–2. It generates a resource bundle the first time that you invoke the Select Text Resource dialog, as illustrated in Figure 32–1.

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.

32.2.3 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 Section 32.2, "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 Section 32.1.2, "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 32–1.

   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**.

32.2.4 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 Section 32.2.2, "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.

32.3 Manually Defining Resource Bundles and Locales

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.

32.3.1 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 Section 32.3, "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 Section 32.1.2, "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 Section 32.3.3, "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**.

32.3.2 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 Section 32.3, "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 Section 32.1.2, "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 Section 32.3.3, "How to Edit a Resource Bundle File." Example 32–5 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 32–5 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"}
      };
}
```

32.3.3 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 Section 32.3, "Manually Defining Resource Bundles and Locales."

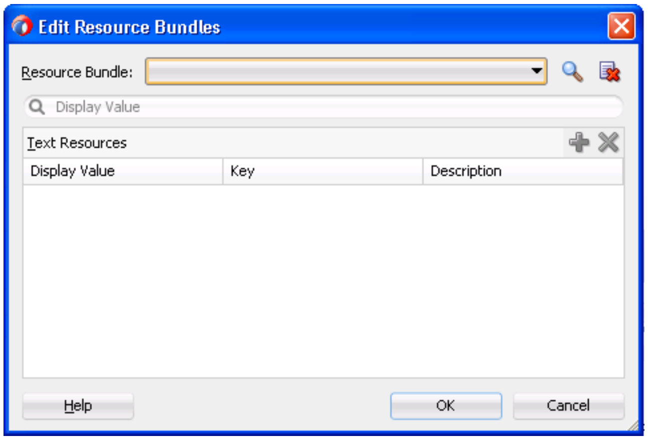
You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Section 32.1.2, "Additional Functionality for Internationalizing and Localizing Pages."

To edit a resource bundle after it has been created:

1. From the main menu, choose **Application > Edit Resource Bundles** from the main menu.

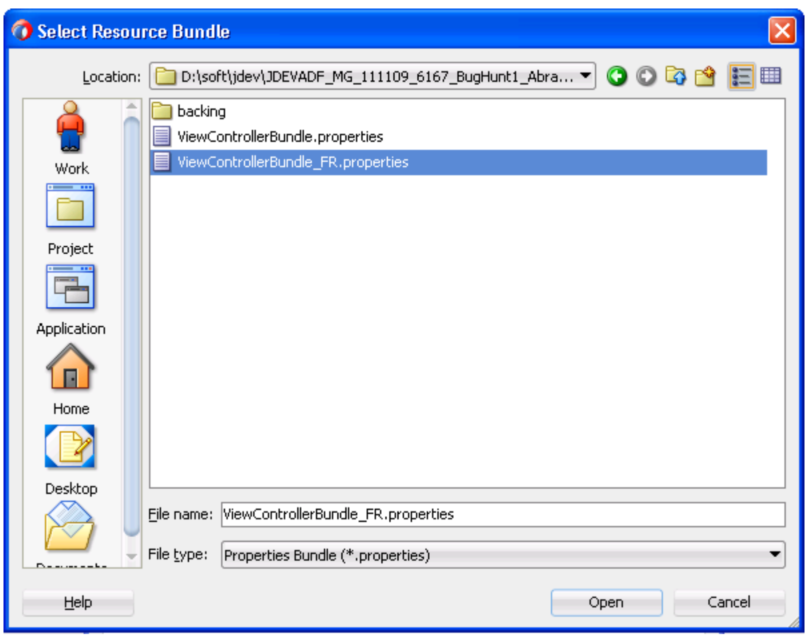
2. In the Edit Resource Bundles dialog, select the resource bundle file you want to edit from the **Resource Bundle** dropdown list, as shown in Figure 32–3, or click the **Search** icon to launch the Select Resource Bundle dialog if the resource bundle file you want to edit does not appear in the **Resource Bundle** dropdown list.

Figure 32–3 Edit Resource Bundle Dialog



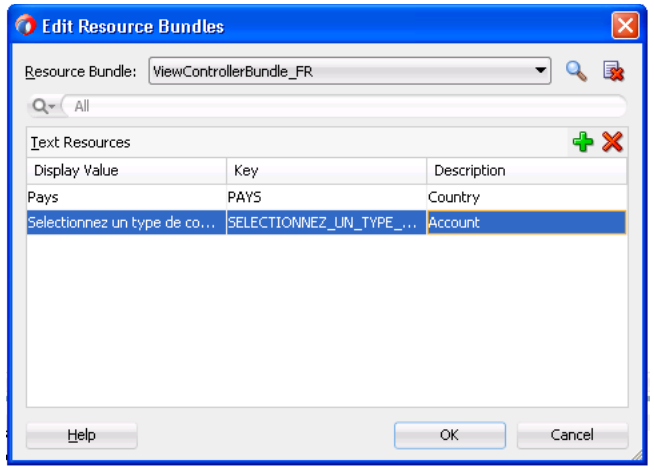
3. In the Select Resource Bundle dialog, select the file type from the **File type** dropdown list. Navigate to the resource bundle you want to edit, as shown in Figure 32–4. Click **OK**.

Figure 32–4 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 32–5. When you have finished, click **OK**.

Figure 32–5 Adding Values to a Resource Bundle



32.3.4 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:

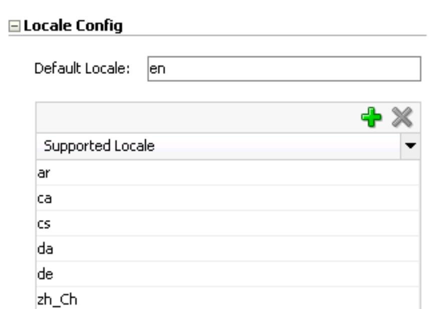
It may help to understand how you can manually manage resource bundles. For more information, see Section 32.3, "Manually Defining Resource Bundles and Locales."

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Section 32.1.2, "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 32–6.

Figure 32–6 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_Ch</supported-locale>
 </locale-config>
```

32.3.5 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:

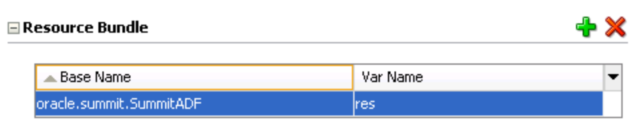
It may help to understand how you can manually manage resource bundles. For more information, see Section 32.3, "Manually Defining Resource Bundles and Locales."

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Section 32.1.2, "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 32–7.

Figure 32–7 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>
```

32.3.6 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 Section 32.3, "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 Section 32.1.2, "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. Example 32–6 shows the encoding for a sample page.

Example 32–6 Page and Response Encoding

```
<?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. Example 32–7 shows the code for the **View** button on the myDemo page.

Example 32–7 Binding to a Resource Bundle

```
<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 32–8 shows how `adfBundle` is used to provide the button text with a resource string from a specific resource bundle.

Example 32–8 Binding Using adfBundle

```
<af:button text="#{adfBundle['project.EmpMsgBundle'] ['Deptno_LABEL']}"
```

32.3.7 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.

32.3.8 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 32–9, to make it overrideable. Once it is marked as overriden, 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.

Example 32–9 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>
```

For more information about the `adf-config.xml` file, see Section A.4, "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*.

32.4 Configuring Pages for an End User to Specify Locale 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.

32.4.1 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 Section 32.4, "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 Section 32.1.2, "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 Section 3.4.1, "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 Example 32–10.

Example 32–10 Component to Choose a Locale

```
<af:selectOneChoice label="Select Locale"
    binding="#{backingBeanScope.backing_changeLocale.soc1}"
    id="soc1">
  <af:selectItem label="French" value="FR"
    binding="#{backingBeanScope.backing_changeLocale.si1}"
    id="si1"/>
    ...
</af:selectOneChoice>
```

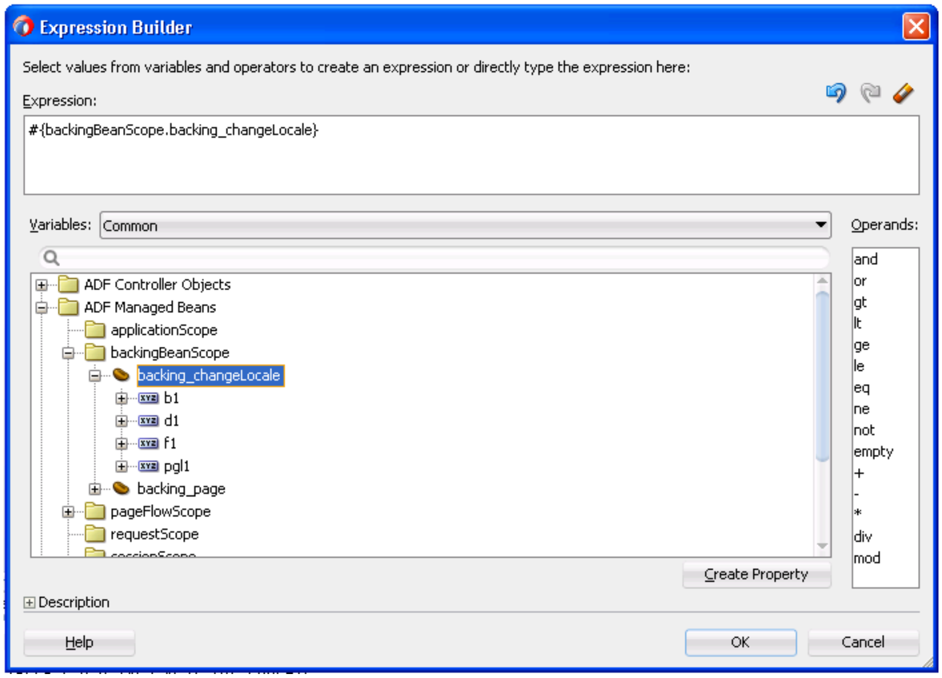
4. Bind the `locale` attribute of the `f:view` tag to the locale value in the backing bean.

   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 32–8.

Figure 32–8 Expression Builder Binding the Locale Attribute to a Backing Bean



5. Save the page.

32.4.2 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 32–11 shows an example using the `selectOneChoice` component.

Example 32–11 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.si1}"
 id="si1"/>
          ...
</af:selectOneChoice>
```

JDeveloper also generates the required methods in the backing bean for the page. Example 32–12 shows extracts for the backing bean that correspond to Example 32–11.

Example 32–12 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 setSi1(RichSelectItem si1) {
        this.si1 = si1;
    }
...
}
```

32.4.3 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 Section 32.3.4, "How to Register a Locale for Your Application."

32.5 Configuring Optional ADF Faces Localization Properties

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 Section A.6, "Configuration in trinidad-config.xml."

32.5.1 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 Section 32.3, "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 Section 32.1.2, "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 32–13 shows a sample `trinidad-config.xml` file with all the optional localization elements set.

Example 32–13 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>
```

33

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:

33.1 About Accessibility Support In ADF Faces

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) 1.0 AA standards. The interpretation of these standards is available at `http://www.oracle.com/accessibility/standards.html`

33.1.1 ADF Faces Accessibility Support Use Cases and Examples

ADF Faces provides three types of application accessibility support: screen reader, high-contrast mode, and large-font mode. Figure 33–1 shows an example of accessibility options in a dropdown menu. The user can choose a combination of all three options as required.

Figure 33–1 Dropdown Menu Listing Accessibility Support Options

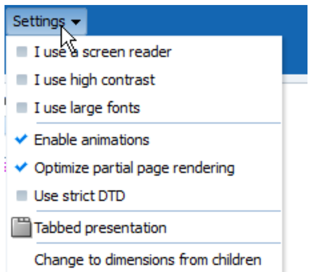
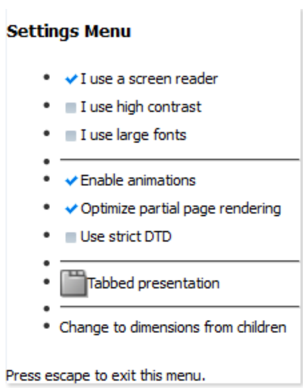


Figure 33–2 shows an example of accessibility options in screen reader mode.

Figure 33–2 Dropdown Menu listing Accessibility Support options in Screen Reader mode



33.1.2 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 Chapter 32, "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 Appendix C, "Keyboard Shortcuts."

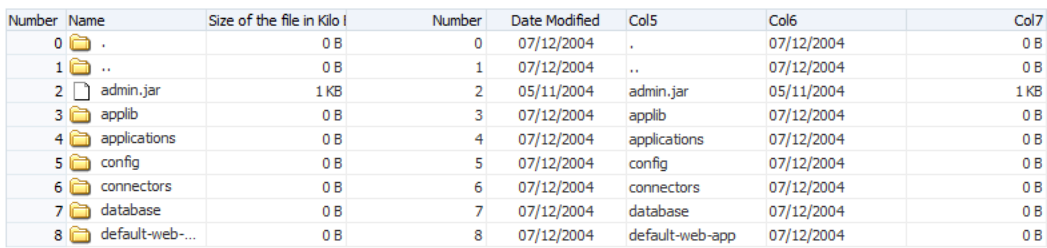
33.2 Configuring Accessibility Support In ADF Faces

ADF Faces provides two levels of application accessibility support, configured in the `trinidad-config.xml` file using the `<accessibility-mode>` element. The acceptable values for `<accessibility-mode>` are:

- `default`: By default, ADF Faces generates components that have rich user interface interaction, and are also accessible through the keyboard. Note that in the default mode, screen readers cannot access all ADF Faces components. If a visually impaired user is using a screen reader, the `screenReader` mode should be used.
- `screenReader`: ADF Faces generates components that are optimized for use with screen readers. The `screenReader` mode facilitates the display for visually impaired users, but will degrade the display for sighted users (without visual impairment).

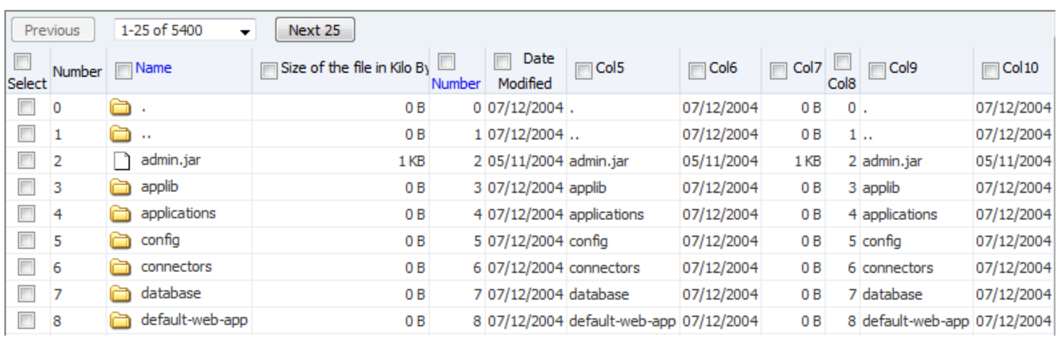
Figure 33–3 shows the table component in the default mode and the screen reader mode.

Figure 33–3 Table Component in Screen Reader Mode



Number	Name	Size of the file in Kilo I	Number	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-...	0 B	8	07/12/2004	default-web-app	07/12/2004	0 B

Screen Reader Disabled



Previous | 1-25 of 5400 | Next 25

Select	Number	Name	Size of the file in Kilo By	Number	Date Modified	Col5	Col6	Col7	Col8	Col9	Col10
	0	.	0 B	0	07/12/2004	.	07/12/2004	0 B	0	.	07/12/2004
	1	..	0 B	1	07/12/2004	..	07/12/2004	0 B	1	..	07/12/2004
	2	admin.jar	1 KB	2	05/11/2004	admin.jar	05/11/2004	1 KB	2	admin.jar	05/11/2004
	3	applib	0 B	3	07/12/2004	applib	07/12/2004	0 B	3	applib	07/12/2004
	4	applications	0 B	4	07/12/2004	applications	07/12/2004	0 B	4	applications	07/12/2004
	5	config	0 B	5	07/12/2004	config	07/12/2004	0 B	5	config	07/12/2004
	6	connectors	0 B	6	07/12/2004	connectors	07/12/2004	0 B	6	connectors	07/12/2004
	7	database	0 B	7	07/12/2004	database	07/12/2004	0 B	7	database	07/12/2004
	8	default-web-app	0 B	8	07/12/2004	default-web-app	07/12/2004	0 B	8	default-web-app	07/12/2004

Screen Reader Enabled

You should provide the ability to switch between the above accessibility support levels in the application, so that users can choose their desired type of accessibility support, if required.

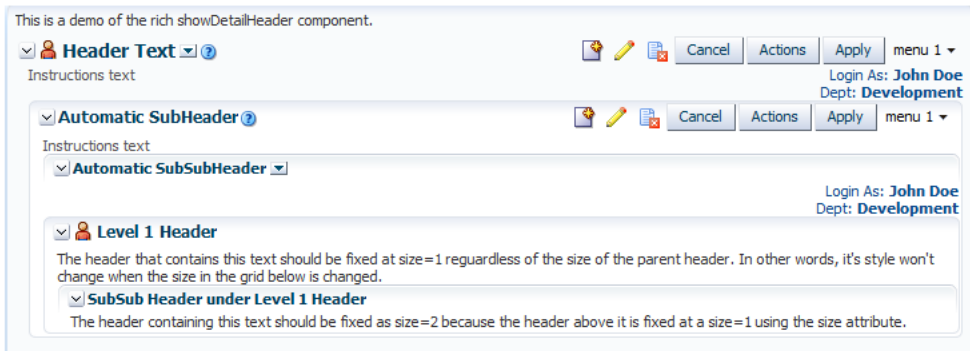
You can also use the `@accessibility-profile` element to define finer-grain accessibility preferences in the style sheet or you can specify the accessibility profile options in the `trinidad-config.xml` file. The options are `high-contrast`, `large-fonts`, or both. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

The acceptable values for `<accessibility-profile>` are:

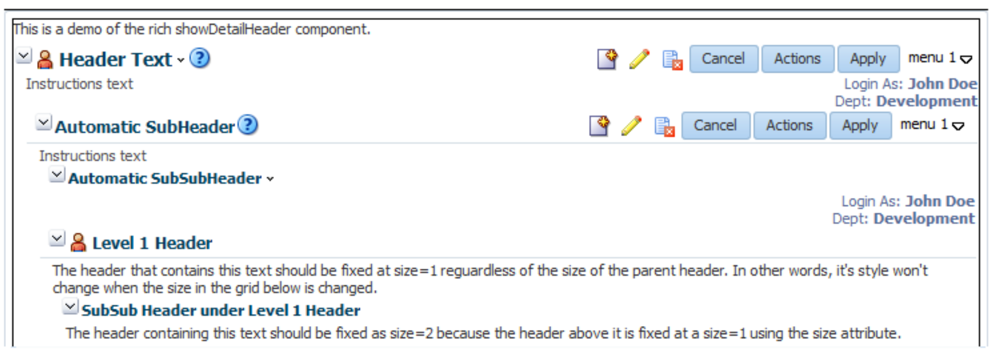
- `high-contrast`: ADF Faces can generate high-contrast–friendly visual content. High-contrast mode is intended to make ADF Faces applications compatible with operating systems or browsers that have high-contrast features enabled. For example, ADF Faces changes its use of background images and background colors in high-contrast mode to prevent the loss of visual information. Note that the ADF Faces high-contrast mode is more beneficial if used in conjunction with your browser's or operating system's high-contrast mode. Also, some users might find it beneficial to use large-font mode along with high-contrast mode.

Figure 33–4 shows the `showDetailHeader` component in high-contrast mode.

Figure 33–4 showDetailHeader Component in High-Contrast Mode



High Contrast Disabled



High Contrast Enabled

- `large-fonts`: ADF Faces can generate browser-zoom–friendly content. In default mode, most text and many containers have a fixed font size to provide a consistent and defined look. In large-font mode, text and containers have a scalable font size. This allows ADF Faces both to be compatible with browsers that are set to larger font sizes and to work with browser-zoom capabilities. Note that if you are not using large-font mode or browser-zoom capabilities, you should disable large-font mode. Also, some users might find it beneficial to use high-contrast mode along with the large-font mode.

  Figure 33–5 shows the `navigationPane` component in large-font mode.

Figure 33–5 navigationPane Component in Large-Font Mode

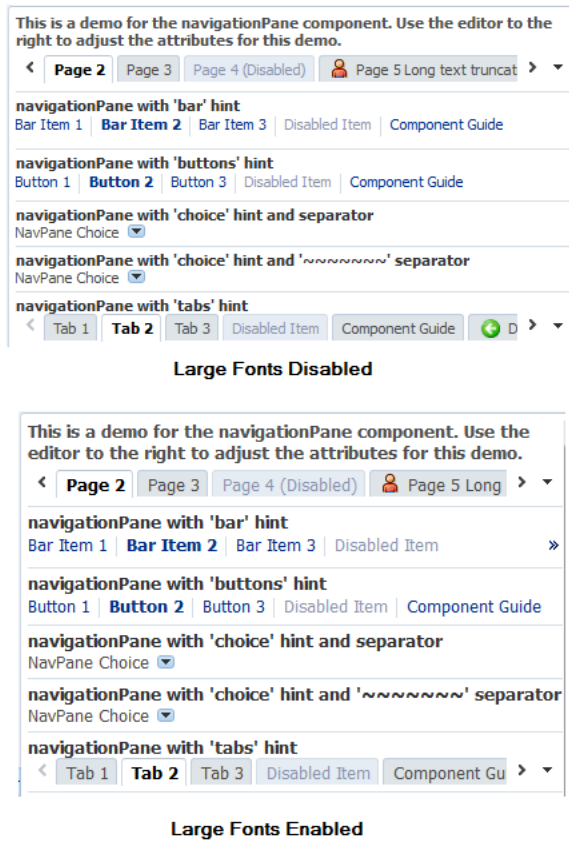


Figure 33–6 shows the font size of text in normal and large-font mode.

Figure 33–6 Text in Normal and Large-Font Mode

Note: The <accessibility-mode> and <accessibility-profile> elements should be EL-bound to a session scope managed bean that contains the user-specific preferences.

33.2.1 Accessibility Support Guidelines at Sign-In

When developing an application, it is a good practice to provide accessibility selection options for users after application sign-in. The sign-in accessibility flow should consist of three pages for first-time users and two pages for subsequent users, as described here:

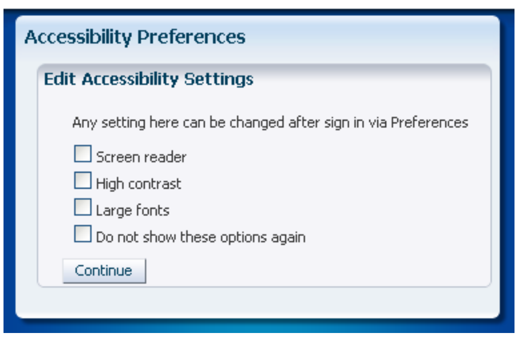
1. Sign-in page
2. Accessibility Mode Selection page (mandatory for first-time users)
3. Application home page

The accessibility options should appear for first time users and any user who opts not to circumvent it on subsequent authentications. For example, you can provide a page or dialog with the following options:

- Hint text: Provide brief information about the accessibility options.
- Checkboxes
  - Screen reader: Generate ADF Faces components that are optimized for use with screen readers.
  - High Contrast: Generate high-contrast–friendly visual content.
  - Large Fonts: Generate browser-zoom–friendly content
  - Do not show these options again: Do not show the accessibility options after sign-in.
- A **Continue** button: Navigate to the home page of the application.

Figure 33–7 shows a page with accessibility options.

Figure 33–7 Accessibility Options After Sign-In



If the user opts not to see the accessibility options at sign-in by choosing `Do not show these options again`, then you should also provide a page or a dialog to navigate the user to the accessibility options. For example, the user preferences page of the application. Figure 33–8 shows the preferences page of an application with accessibility options.

Figure 33–8 Accessibility Options in User Preferences Page

Note that the application may have additional authentication security set up between the Accessibility Mode Selection page and the product-specific home page.

33.2.2 How to Configure Accessibility Support 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 file has a simple XML structure that enables you to define element properties using the JSF expression language (EL) or static values. The order of elements in the file does not matter. You can configure accessibility support by editing the XML file directly or by using the Structure window.

Before you begin:

It may be helpful to have an understanding of accessibility support in ADF Faces. For more information, see Section 33.2, "Configuring Accessibility Support In ADF Faces." You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 33.1.2, "Additional Information for Accessibility Support in ADF Pages."

To configure accessibility support in trinidad-config.xml:

1. In the Applications window, double-click **trinidad-config.xml**.
2. In the XML editor, enter the element name `<accessibility-mode>` and accessibility support value (`default`, `screenReader`, or `inaccessible`). For example:

   ```
   <accessibility-mode>screenReader</accessibility-mode>
   ```

   This code sets the application's accessibility support to screen reader mode.
3. Enter the element name `<accessibility-profile>` and accessibility profile value (`high-contrast`, `large-fonts`).

   For example:

   ```
   <!-- Enable both high-contrast and large-fonts content -->
   <accessibility-profile>high-contrast large-fonts</accessibility-profile>
   ```

   This code sets the application's profile support to use both `high-contrast` and `large-fonts`. Figure 33–9 illustrates a **`trinidad-config.xml`** file in JDeveloper.

Figure 33–9 trinidad-config.xml in JDeveloper

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved. -->
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config" >
  <!-- Enable debug output -->
  <debug-output>false</debug-output>
  <!-- We define some custom styles in the richdemo skin -->
  <!-- We want to use 'demoComponents' skin defined in trinidad-skins.xml
       when running the component skinning demos  -->
  <skin-family>#{demoSkin.skinFamily}</skin-family>
  <skin-version>#{demoSkin.skinVersion}</skin-version>

  <!-- Screen reader mode can be set on the index page -->
  <accessibility-mode>screenReader</accessibility-mode>

  <!-- Screen reader mode can be set on the index page -->
  <accessibility-profile>high-contrast large-fonts</accessibility-profile>
</trinidad-config>
```

Alternatively, you can use the Structure window to insert the value:

1. In the Applications window, select the **trinidad-config.xml** file.
2. In the Structure window, right-click the XML file root element, choose the **Insert Inside trinidad-config > Browse** menu item.
3. In the Insert Item dialog, and select the element name (**accessibility-mode** or **accessibility-profile**) and click **OK**.
4. With the newly inserted element selected in the Structure window, open the Properties window, and enter a value or select one from the dropdown list.

Once you have configured the `trinidad-config.xml` file, you can retrieve the property values programmatically or by using JSF EL expressions.

For example, the following code returns nothing if the accessibility mode is not explicitly set:

```
String mode=ADFFacesContext.getCurrentInstance().getAccessibilityMode;
```

In this EL expression example, a null value is returned if the accessibility mode is not explicitly set:

```
<af:outputText value="*#{requestContext.accessibilityMode}"/>
```

You can set accessibility selection options at the time of application sign-in, or in the user preferences page of the application. For more information, see Section 33.2.1, "Accessibility Support Guidelines at Sign-In."

33.3 Specifying Component-Level Accessibility Properties

Guidelines for component-specific accessibility are provided in Section 33.3.1, "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 Section 33.5, "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 `&amp` 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 Section 33.4, "Creating Accessible Pages."

33.3.1 ADF Faces Component Accessibility Guidelines

To develop accessible ADF Faces components, follow the guidelines described in the component's tag documentation, and in Table 33–1. Components not listed in Table 33–1 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 33–1 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 atleast 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. In rich mode, the label is not displayed, but it is read by the screen reader software in screen reader mode. 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.

Table 33–1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
`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` `af:document`	Specify the `title` property. If you wish to provide help information, use `helpTopicId`.
`af:goMenuItem`	Specify the `text` 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. Usually, the `text` or `textAndAccessKey` property is used as the label for the component. Unique buttons and links must have unique text.
`af:icon`	Specify the `shortDesc` property. If the icon is only present only for decorative purposes and communicates no information, set `shortDesc` to the empty string.
`af:image`	Specify the `shortDesc` property. If the image is only present for decorative purposes and communicates no information, set `shortDesc` to the empty string. Use the `longDescURL` 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.
`af:inlineFrame`	Specify the `shortDesc` property.

`af:inputColor`
`af:inputComboboxListOfValues`
`af:inputDate`
`af:inputFile`
`af:inputListOfValues`
`af:inputNumberSlider`
`af:inputNumberSpinbox`
`af:inputRangeSlider`
`af:inputText`
	Specify the `label` property.
	For `af:inputComboboxListOfValues` and `af:inputListOfValues` components, the `searchDesc` must also be specified.
	If you wish to provide help information, use `helpTopicId`.

`af:outputFormatted`	The `value` property must specify valid HTML.
`af:outputLabel`	Specify the `value` or `valueAndAccessKey` property.
`af:panelBox` `af:panelHeader`	Specify the `text` property. 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` `af:panelStretchLayout`	Refer to Section 33.4.4, "How to Use Page Structures and Navigation."

Table 33–1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
`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: ■ `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` `af:selectBooleanRadio`	One of the following properties must be specified: `text`, `textAndAccessKey`, or `label`. 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.

`af:selectManyCheckbox`
`af:selectManyChoice`
`af:selectManyListbox`
`af:selectManyShuttle`
`af:selectOneChoice`
`af:selectOneListbox`
`af:selectOneRadio`
`af:selectOrderShuttle`
	Specify the `label` property.
	For the `af:selectManyShuttle` and `af:selectOrderShuttle` components, the `leadingHeader` and `trailingHeader` properties must be specified.
	If you wish to provide help information, use `helpTopicId`.

`af:showDetailHeader`	Specify the `text` property. If you wish to provide help information, use `helpTopicId`.
`af:showDetailItem`	One of the following properties must be specified: `text`, `textAndAccessKey`, or `shortDesc`.

Table 33–1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
`af:showPopupBehavior`	Specify the `triggerType` property. Note that trigger types of `mouseHover`, `mouseMove`, `mouseOver`, and `mouseOut` are suppressed in the screen reader mode. Hence, you must provide alternate access to the content for these trigger types in the screen reader mode.
`af:table` `af:treeTable`	Specify the `summary` property. The summary should describe the purpose of the table. All table columns must have column headers. In screen reader mode, if a label is specified for an input component inside `af:column`, 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.

33.3.2 Using ADF Faces Table Components in Screen Reader Mode

If you are using ADF Faces table components in your web application, you must designate a column as the row header for screen reader mode. The row header is used by the screen reader software to announce the row when the end user selects it. Typically, a single column is used as a row header that allows multiple selections, but you can mark multiple columns as row headers. When you mark multiple columns as row headers, they appear as the initial columns of the table, and they are frozen.

Sometimes, for display purposes, you may not want to have a row header. In such a case, you must define one column in the table to have the `rowHeader` attribute set to `unstyled`. In screen reader mode, the table or the tree table component with the `unstyled` row header column is moved to the starting position with `displayIndex` set to `0`, and it is frozen. In default mode, the table or tree table component with the `unstyled` row header column is not moved to the starting position, it is not frozen, and it is rendered without any row header CSS style.

33.3.3 ADF Data Visualization Components Accessibility Guidelines

To develop accessible ADF Data Visualization Components, follow the accessibility guidelines described in Table 33–2. Components not listed do not have accessibility guidelines.

Table 33–2 ADF Data Visualization Components Accessibility Guidelines

Component	Guideline

`dvt:projectGantt`
`dvt:resourceUtilization Gantt`
`dvt:schedulingGantt`
	Specify the `summary` property. The summary should describe the purpose of the Gantt chart component.

`dvt:gauge`	Specify the `shortDesc` property. The property should describe the purpose of the gauge.

Table 33–2 (Cont.) ADF Data Visualization Components Accessibility Guidelines

Component	Guideline

`dvt:areaGraph`
`dvt:barGraph`
`dvt:horizontalBarGraph`
`dvt:bubbleGraph`
`dvt:comboGraph`
`dvt:funnelGraph`
`dvt:lineGraph`
`dvt:paretoGraph`
`dvt:pieGraph`
`dvt:radarGraph`
`dvt:scatterGraph`
`dvt:stockGraph`
	Specify the `shortDesc` property. The `shortDesc` property should describe the purpose of the graph.
	Note that in screen reader mode, an instance of pivot table substitutes for the graph component, and the end user can then use the standard cursor keys to navigate through the data.
	In screen reader mode, the following visualization features of the graph component are not supported:
	■ Data change animation during partial page rendering.
	■ Zoom and scroll. Scrolling is supported in pivot table.
	■ The `seriesRolloverBehavior` and `hideAndShowBehavior` properties on simple graph tags.
	■ The `interactiveSliceBehavior` property on pie graphs.
	■ Precise control of data marker shapes and colors, including the following:
	  ■ Declarative properties on the `Series` child tag
	  ■ Declarative `markerShape` and `markerColor` properties on scatter graphs
	  ■ Callback APIs
	  ■ Conditional formatting rules from a backing bean
	  ■ Marker underlays for bubble and scatter graphs
	In screen reader mode, the following interactive features of the graph component are not supported:
	■ Context menu facets
	■ Popups
	■ `TimeSelector` functionality through the `<dvt:timeSelector>` child tag
	■ The `drillingEnabled` property of simple graph tags
	■ `ShapeAttributes` support, and access to fine-grained mouse and key events from all graph components
	■ Drag and drop in bubble and scatter graphs
	■ `DataSelection` in bubble and scatter graphs
	■ Programmatic `TickLabelCallback` support

`dvt:hierarchyViewer`
`dvt:sunburst`
`dvt:treemap`
	Specify the `summary` property.
	Note that in screen reader mode, an instance of the tree table component substitutes for the component, and the end user can then use the standard cursor keys to navigate through the data.

`dvt:map`	Specify the `summary` property. Note that in screen reader mode, an instance of the table component substitutes for the geographic map component, and the end user can then use the standard cursor keys to navigate through the data.

Table 33–2 (Cont.) ADF Data Visualization Components Accessibility Guidelines

Component	Guideline
`dvt:thematicMap`	Specify the `summary` property. Note that in screen reader mode, an instance of the table component substitutes for each `DataLayer` component, and the end user can then use the standard cursor keys to navigate through the data. If the thematic map instance has multiple `DataLayers` associated with it, then a dropdown list is also rendered in screen reader mode to enable end users to switch between the corresponding table instances.
`dvt:pivotTable`	Specify the `summary` property. The summary should describe the purpose of the timeline component.
`dvt:timeline`	Specify the `summary` property. The summary should describe the purpose of the pivot table component.
`dvt:sparkChart`	Specify the `shortDesc` property.

33.3.4 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 Section 33.3, "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 Section 33.1.2, "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.

  Example 33–1 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.

Example 33–1 AccessKey Attribute Defined

```
<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.

Example 33–2 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.

Example 33–2 TextAndAccessKey Attribute Defined

```
<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.

  Example 33–3 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.

Example 33–3 LabelAndAccessKey Attribute Defined

```
<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.

  Example 33–4 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.

Example 33–4 ValueAndAccessKey Attribute Defined

```
<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.

33.3.5 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 `f: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 Section 33.3, "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 Section 33.1.2, "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 Section 32.3.1, "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 Section 32.3.4, "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 amp) 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&ggiorna
   ```

4. Add a <resource-bundle> element to the faces-config.xml file for your application. Example 33–5 shows an entry in a JSF configuration file for a resource bundle.

Example 33–5 Resource Bundle in JSF Configuration File

```
<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 Chapter 32, "Internationalizing and Localizing Pages."

33.4 Creating Accessible Pages

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 33–3.

Table 33–3 General Accessibility Guidelines

Guideline	Action
Avoid using raw HTML content	If possible, avoid using raw HTML content. If raw HTML content is required, use `af:outputFormatted` 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.
Provide access to the accessibility mode	ADF Faces exposes the following accessibility modes: ■ Default ■ Screen reader - optimized for screen reader users You must design your web application to enable end users to choose a screen reader mode, if required. For more information about configuring screen reader mode in ADF Faces, see Section 33.2, "Configuring Accessibility Support In ADF Faces." When designing your web application, note that you may be required to add additional accessibility modes, such as a high-contrast accessibility mode or a large-font mode.
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.

33.4.1 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 33–4.

Table 33–4 Partial Page Rendering Guidelines for Accessibility

Guideline	Action
Prefer downstream partial page changes	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. 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.
Provide guidance for partial page changes	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.

33.4.2 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 33–5.

Table 33–5 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.
Do not use JavaScript timeouts	Screen readers do not reliably track modifications made in response to timeouts implemented using the JavaScript `setTimeout()` or `setInterval()` APIs. Do not call these methods.

Table 33–5 (Cont.) Scripting Guidelines for Accessibility

Guideline	Action
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: ■ 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 when in screen reader mode, 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 `OBJECT` element must specify the `title` attribute; the `APPLET` element must specify the `alt` attribute. Run the audit report to verify the audit rule for `af:media`.
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 33–6.

Table 33–6 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 `DOCTYPE` of each page	No action required. ADF Faces specifies the `DOCTYPE` for every page.
Avoid using ASCII characters to render drawings or figures	Ensure that no ASCII art is included in the application.

Table 33–6 (Cont.) Application Programming Guidelines for Accessibility

Guideline	Action
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 depreciated 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 `H1` through `H6`	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 `OL`, `UL`, `LI`, `DL`, `DT`, and `DD`.
Mark quotations with proper elements	Ensure that quotations are appropriately marked up using `Q` or `BLOCKQUOTE` elements. Do not use quotation markup for formatting effects such as indentation.
Identify the primary natural language of each page with the `lang` attribute on the HTML element	No action required. ADF Faces renders the `lang` attribute on every page.
Ensure that all form elements have a label associated with them using markup	Run the audit report. The `Verify that the component is labeled` audit rule warns about missing labels.
Provide unique titles to each `FRAME` or `IFRAME` elements	Run the audit report. The `Verify that the component has a short description` audit rule warns when `af:inlineFrame` is missing the `shortDesc` title. Note that ADF Faces `af:inlineFrame` does not provide access to `longDesc`.
Provide a title to each page of the frame	Run the audit report. The `Verify that the component has a title` audit rule warns when `af:document` is missing the `title` 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 `af:clientListener` or `af:showPopupBehavior`, you must ensure that the pop-up window is launched in response to explicit user action.

33.4.3 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 33–7.

Table 33–7 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 `em`, `ex` or `%` units rather than `px`. 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. It is important to do this in large fonts mode, which is the mode that is best compatible with browser zoom.
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.

33.4.4 How to Use Page Structures and Navigation

When using page structures and navigation tools, you must follow the guidelines as described in Table 33–8.

Table 33–8 Style Guidelines for Page Structures and Navigation

<table>
<tr><th>Guideline</th><th>Action</th></tr>
<tr><td>Use <code>af:panelSplitter</code> for layouts</td><td>When implementing geometry-managed layouts, using <code>af:panelSplitter</code> allows users to:
Redistribute space to meet their needs
Hide or collapse content that is not of immediate interest. If you are planning to use <code>af:panelStretchLayout</code>, you should consider using <code>af:panelStretchLayout</code> instead when appropriate. These page structure qualities are useful to all users, and are particularly helpful for low-vision users and screen-reader users As an example, a chrome navigation bar at the top of the page should be placed within the <code>first</code> facet of a vertical <code>af:panelSplitter</code> component, rather than within the <code>top</code> 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. </td></tr>
<tr><td>Enable scrolling of flow layout contents</td><td>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.</td></tr>
<tr><td>Use header based components to identify page structure</td><td>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:
<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)</td></tr>
<tr><td>Use <code>af:breadCrumbs</code> to identify page location</td><td>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.</td></tr>
<tr><td>Use af:skipLinkTarget to provide a skip link target</td><td>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. </td></tr>
<tr><td>Maintain consistency for navigational mechanisms that are repeated on multiple pages</td><td>Ensure navigation consistency by using the ADF Faces navigation components.</td></tr>
<tr><td>Provide a method for skipping repetitive content</td><td>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.</td></tr>
</table>

33.4.5 How to Use Images and Tables

When using images, you must follow the guidelines as described in Table 33–9.

Table 33–9 Style Guidelines for Images

Guideline	Action
Specify description in `alt` attribute of non-decorative images	Run the audit report. The `Verify that the component has a short description audit rule` warns about missing `shortDesc` attributes. Ensure that `shortDesc` value is meaningful.
Ensure that decorative images, such as spacer images, specify an `alt=""` attribute	Run the audit report. The `Verify that the component has a short description audit rule` warns about missing `shortDesc` attributes. Ensure that `shortDesc` value is meaningful.
Specify description in `alt` attribute of complex images, such as charts	Ensure that the `longDesc` attribute is specified for complex `af:image` components. You may consider replacing charts with an accessible component, such as a table, in screen reader mode.
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 33–10.

Table 33–10 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 `Verify that table columns have headers` audit rule warns when column header data is missing. Applications which use `trh:tableLayout` to construct data or layout tables are responsible for ensuring that such tables adhere to all Oracle accessibility guidelines.
Provide a description for each table component using the `summary` attribute or `CAPTION` element.	Run the audit report. The `Verify that tables has summaries` audit rule warns when data tables are missing the summary attribute.
Ensure that layout tables do not use the `TH` element.	No action required. ADF Faces ensures that layout components do not use `TH` for layout tables.
Ensure that layout tables specify `summary=""` and do not have the `CAPTION` 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.

33.4.6 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 33–11.

Table 33–11 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.

33.5 Running Accessibility Audit Rules

JDeveloper provides ADF Faces accessibility audit rules to investigate and report compliance with many of the common requirements described in Section 33.3.1, "ADF Faces Component Accessibility Guidelines."

33.5.1 How to Create an Audit Profile

You can create an audit profile from the Preferences dialog.

Before you begin:

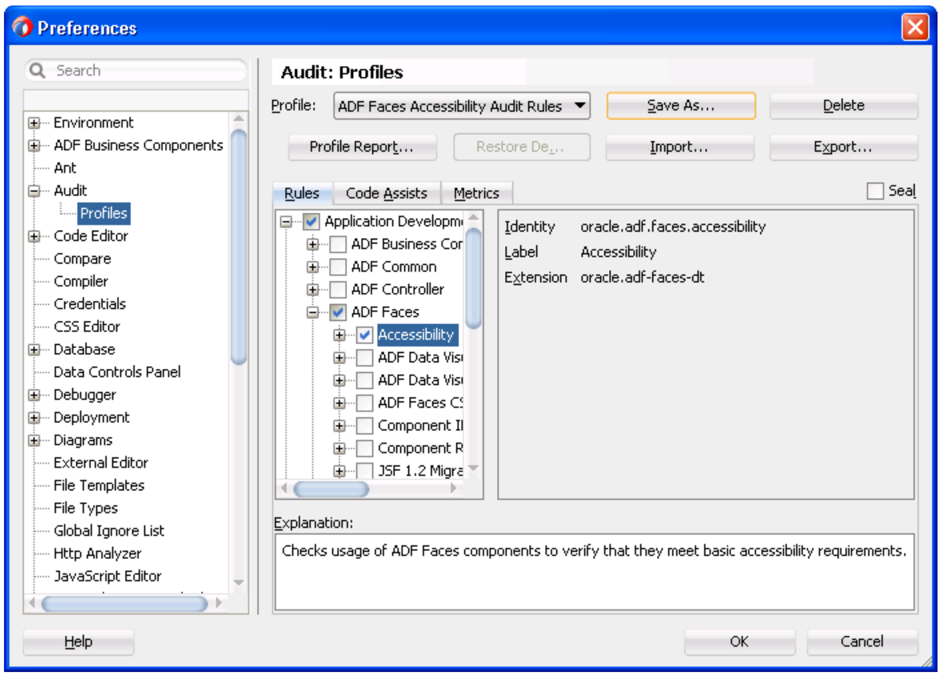
It may be helpful to have an understanding of accessibility audit rules. For more information, see Section 33.5, "Running Accessibility Audit Rules." You may also find it helpful to understand functionality that can be added using other ADF Faces features. For more information, see Section 33.1.2, "Additional 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 33–10 illustrates the settings of the Audit: Profiles dialog to create an accessibility audit profile.

Figure 33–10 Audit Profile Settings for ADF Faces Accessibility



5. Click **OK**.

33.5.2 How to Run Audit Report

Running an audit report requires creating and running an audit profile.

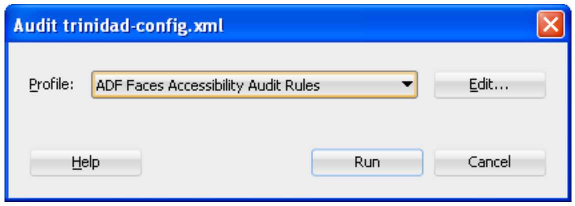
Before you begin:

It may be helpful to have an understanding of accessibility audit rules. For more information, see Section 33.5, "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 Section 33.1.2, "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 33–11 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.

34

Creating Custom ADF Faces Components

This chapter describes how to create custom ADF Faces components.

This chapter includes the following sections:

- Section 34.1, "About Custom ADF Faces Components"
- Section 34.2, "Setting Up the Workspace and Starter Files"
- Section 34.3, "Developing for the Client-Side"
- Section 34.4, "Developing for the Server-Side"
- Section 34.5, "Deploying a Component Library"
- Section 34.6, "Adding the Custom Component to an Application"

34.1 About Custom ADF Faces Components

The ADF Faces component library provides a comprehensive set of UI components that covers most of your requirements. However, there are situations when you will want to create a custom rich component that is specific to your application. A custom rich component will allow you to have custom behavior and perform actions that best suit the needs of your application.

Note: Creating custom JSF Facelet components is covered in many books, articles, web sites, and the JavaServer Faces specification, therefore, it is not covered in this guide. This chapter describes how to create ADF Faces components.

JSF technology is built to allow self-registering components and other framework parts. The core JSF runtime at web application startup accomplishes this by inspecting all JAR files in the class path. Any JAR files whose `/META-INF/faces-config.xml` file contains JSF artifacts will be loaded. Therefore, you can package custom ADF Faces components in a JAR file and simply add it into the web project.

For each ADF Faces component, there is a server-side component and there can also be a client-side component. On the server, for JSPs, a render kit provides a base to balance the complex mixture of markup language and JavaScript. The server-side framework also adds a custom lifecycle to take advantage of the API hooks for partial page component rendering. On the client, ADF Faces provides a structured JavaScript framework for handling various nontrivial tasks. These tasks include state synchronization using partial page rendering. For more information about the ADF Faces architecture, see Chapter 4, "Using ADF Faces Client-Side Architecture."

ADF Faces components are derived from the Apache MyFaces Trinidad component library. Because of this, many of the classes you extend when creating a custom ADF Faces component are actually MyFaces Trinidad classes. For more information about the history of ADF Faces, including its evolution, see Chapter 1, "Introduction to ADF Faces."

Between the JSP and the JSF components is the `Application` class. The tag library uses a factory method on the `application` object to instantiate a concrete component instance using the mnemonic referred to as the `componentType`.

A component can render its own markup but this is not considered to be a best practice. The preferred approach is to define a render kit that focuses on a strategy for rendering the presentation. The component uses a factory method on the render kit to get the renderer associated with the particular component. If the component is consumed in an application that uses Facelets, then a component handler creates the component.

In addition to functionality, any custom component you create must use an ADF Faces skin to be able to be displayed properly with other ADF Faces components. To use a skin, you must create and register the skinning keys and properties for your component. This chapter describes only how to create and register skins for custom components. For more information about how skins are used and created in general, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

> **Tip:** To work with ADF Faces components, your custom component must use at least the ADF Faces `simple` skin, because the `skyros` skin inherits from the simple skin. Additionally, if there is any chance your component will be used in an Oracle WebCenter Portal application, then your skin must also be registered with the `simple.portlet` skin.

34.1.1 Developing a Custom Component with JDeveloper

An ADF Faces component consists of both client-side and server-side resources. On the client side, there is the client component, the component peer (the component presenter), and any events associated with the client component.

On the server side, there is the server component, server component events, and event listeners. Also, there is a component renderer, a component JSP tag, a composite resource loader, a JavaScript resource loader, and a resource bundle.

The component also has several configuration and support files. Together, these classes, JavaScripts, and configuration files are packaged into a JAR file, which can be imported as a library into an application and used like other components.

You can use JDeveloper to set up the application workspace and project in which you develop the custom component. After you have created the workspace and project, you add starter working files for the required classes, JavaScript files, and configuration files that make up the custom component. During development, you edit and add code to each of these files, specific for the custom component.

The development process is as follows:

1. Create an application, workspace, and project as an environment for development. This includes adding library dependencies and registering XML schemas. You should not create the component in the same application in which you plan to use the component.
2. Create a deployment profile for packaging the component into a JAR file.
3. Create the following starter configuration and support files:

  - `faces-config.xml`: Used to register many of the artifacts used by the component.
  - `trinidad-skins.xml`: Used to register the skins that the component uses.
  - Cascading style sheet: Used to define the style properties for the skins.
  - Render kit resource loader: Allows the application to load all the resources required by the component.
  - `adf-js-features.xml`: Allows the component to become part of a JavaScript partition. For more information about partitions, see Section 4.9, "JavaScript Library Partitioning."
  - JSP tag library descriptor (TLD) (for JSP): Defines the tag used on the JSF page.
  - Component handler (for Facelets): Defines the handler used to render the component.

4. Create the following client-side JavaScript files:
   - Client Component: Represents the component and its attributes on the client.
   - Client Peer: Manages the document object model (DOM) for the component.
   - Client Event: Invokes processing on the client and optionally propagates processing to the server.
5. Create the following server-side Java files:
   - Server Component class: Represents the component on the server.
   - Server Event Listener class: Listens for and responds to events.
   - Server Events class: Invokes events on the server.
   - Server Renderer class: Determines the display of the component.
   - Resource Bundle class: Defines text strings used by the component.
6. Further develop the component by testing and debugging the JavaScript and Java code. You can use the JDeveloper debugger to set breakpoints and to step through the code. You can also use Java logging features to trace the execution of the component.
7. Deploy the component into a JAR file.
8. Test the component by adding it into an application.

Table 34–1 lists the client-side and server-side component artifacts for a custom component. The configuration and support files are not included in the table.

Table 34–1 Client-Side and Server-Side Artifacts for a Custom Component

Client	Server

Component class:
`oracle.component_ package.js.component.prefixComponent_name.js`
Extends:
`oracle.adf.view.js.component.`
`AdfUIObject.js`
	Component:
	`oracle.<component_ package>.faces.component.<Component_name>.java`
	Extends:
	`org.apache.myfaces.trinidad.component.`
	`UIXObject.java`

Event:
`oracle.<component_ package>.js.event.<prefix><Event_name>.js`
Extends:
`oracle.adf.view.js.component.`
`AdfComponentEvent.js`
	Event:
	`oracle.<component_package>.faces.event.<Event_name> .java`
	Extends:
	`javax.faces.event.FacesEvent.java`

	Event Listener:
	`oracle.<component_package>.faces.event<Listener_ name>`
	Extends:
	`com.faces.event.FacesListener`

Component Peer:
`com.<component_ package>.js.component.<prefix><Peer_ name>Peer.js`
Extends:
`oracle.adf.view.js.laf.rich.`
`AdfRichUIPeer.js.js`	

	Component Renderer:
	`com.<component_package>.faces.render.<Renderer_ name>.java`
	Extends:
	`oracle.adf.view.rich.render.RichRenderer.`
	`java`

	Component JSP Tag (JSP only):
	`com.<component_package>.faces.taglib.<Tagname_ name>Tag.java`
	Extends:
	`javax.faces.webapp.UIComponentELTag.java`

Table 34–1 (Cont.) Client-Side and Server-Side Artifacts for a Custom Component

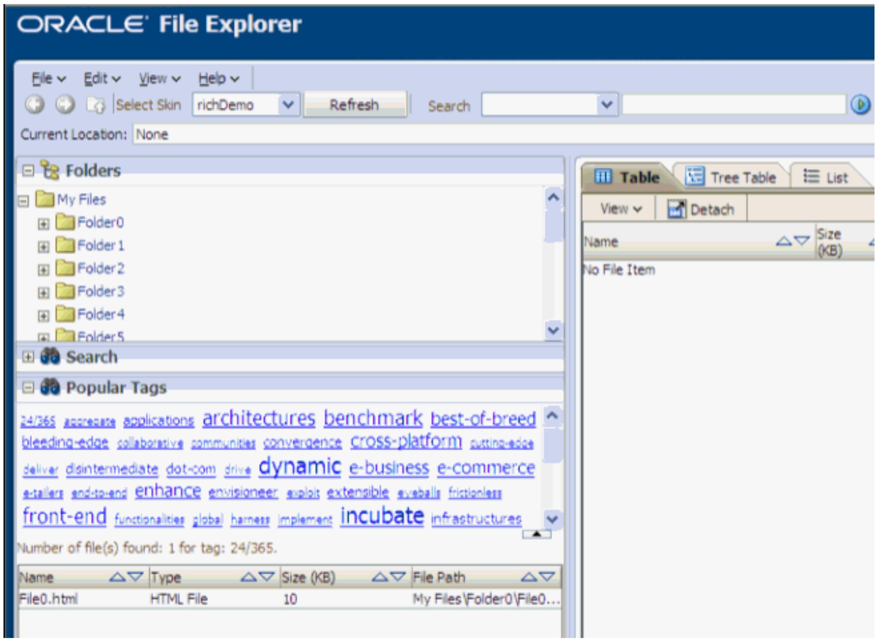
Client	Server
	Composite Resource Loader: com.<component_package>.faces.resource.<Loader_name>ResourceLoader.java **Extends**: org.myfaces.trinidad.resource. RegxResourceLoader.java
	JavaScript Resource Loader: com.<component_package>.faces.resource.<Script_Loader_name>ResourceLoader.java **Extends**: org.myfaces.trinidad.resource. AggregateingResourceLoader.java
	Resource Bundle: com.<component_package>.faces.resource.<Bundle_name>Bundle.java **Extends:** java.util.ListResouceBundle.java

34.1.2 An Example Custom Component

To help illustrate creating a custom component, a custom component named `tagPane` will be used as an example throughout the procedures. The `tagPane` custom component is created for reuse purposes. Although the `tagPane` presentation might have been implemented using a variety of existing components, having a single custom component simplifies the work of the page developer. In this case, there may be a trade-off of productivity between the component developer and the page developers. If this particular view composition were needed more than once, the development team would reduce costs by reducing the lines of code and simplifying the task of automating a business process.

The `tagPane` component displays a series of tags and their weighted occurrences for a set of files. Tags that are most frequently used are displayed in the largest font size, while the least used tags are displayed in the smallest font size. Each tag is also a link that triggers an event, which is then propagated to the server. The server causes all the files that contain an occurrence of that tag to then be displayed in a table. Figure 34–1 shows how the `tagPane` component would be displayed if it was added below the Search pane in the File Explorer application.

Figure 34–1 Custom tagPane Component



The `tagPane` component receives a collection of tags in a Java `Map` collection. The key of the map is the tag name. The value is a weight assigned to the tag. In the File Explorer application, the weight is the number of times the tag occurs and in most cases, the number of files associated with the tag. The tag name is displayed in the body text of a link and the font size used to display the name represents the weight. Each tag's font size will be proportionally calculated within the minimum and maximum font sizes based upon the upper and lower weights assigned to all tags in the set of files. To perform these functions, the `tagPane` custom component must have both client-side and server-side behaviors.

On the server side, the component displays the map of tags by rendering HTML hyperlinks. The basic markup rendering is performed on the server. A custom event on the component is defined to handle the user clicking a link, and then to display the associated files. These server-side behaviors are defined using a value expression and a method expression.

For example, the `tagPane` component includes:

- A `tag` property for setting a `Map<String, Number>` collection of tags.
- A `tagSelectionListener` method-binding event that is invoked on the server when the user clicks the link for the tag.
- An `orderBy` property for displaying the sequence of tags from left to right in the order of descending by weight or alternatively displaying the tag links ascending alphabetically.

To allow each tag to be displayed in a font size that is proportional to its weight (occurrences), the font size is controlled using an inline style. However, each tag and the component's root markup node also uses a style class.

Example 34–1 shows how the `tagPane` component might be used in a JSF page.

Example 34–1 tagPane Custom Component Tag in a JSF Page

```
<acme:tagPane id="tagPane" tags="#{explorer.navigatorManager.tagNavigator.tags}"
  tagSelectListener="#{explorer.navigatorManager.tagNavigator.onTagSelect}"
```

```
  orderBy="alpha"
  partialTriggers="tagCountLabel"/>
```

Because the `tagPane` component must be used with other ADF Faces components, it must use the same skins. Therefore, any styling is achieved through the use of cascading style sheets (CSS) and corresponding skin selectors. For example, the `tagPane` component needs skin selectors to specify the root element, and to define the style for the container of the links and the way the hyperlinks are displayed. Example 34–2 shows a sample set of style selectors in the CSS file for the `tagPane` component.

Example 34–2 CSS Style Selectors for the Sample Custom Component

```
acme|tagPane     - root element
acme|tagPane::content - container for the links
acme|tagPane::tag - tag hyperlink
```

You may need to specify the HTML code required for the custom component on the server side.

Example 34–3 shows HTML server-side code used for the `tagPane` component.

Example 34–3 HTML Code for the Server Side

```
<div class=" acme|tagPane">
  <span class=" acme|tagPane::content ">
    <a class=" acme|tagPane::tag" href="#" style="font-size:9px;">Tag1</a>
    <a class=" acme|tagPane::tag" href="#" style="font-size:10px;">Tag2</a>
  </span>
</div>
```

On the client side, the component requires a JavaScript component counterpart and a component peer that defines client-side behavior. All DOM interaction goes through the peer (for more information, see Chapter 4, "Using ADF Faces Client-Side Architecture"). The component peer listens for the user clicking over the hyperlinks that surround the tag names. When the links are clicked, the peer raises a custom event on the client side, which propagates the event to the server side for further processing.

Table 34–2 lists the client-side and server-side artifacts for the `tagPane` component. Referencing the naming conventions in Table 34–1, the *`component_package`* is `com.adfdemo.acme` and the *`prefix`* is `Acme`.

Table 34–2 Client-Side and Server-Side Artifacts for the tagPane Custom Component

Client	Server

Component:
`com.adfdemo.acme.js.component.`
`AcmeTagPane.js`
Extends:
`oracle.adf.view.js.component.`
`AdfUIObject.js`
	Component
	`com.adfdemo.acme.faces.component.TagPane.java`
	Extends:
	`org.apache.myfaces.trinidad.component.`
	`UIXObject.java`

Event:
`com.adfdemo.acme.js.event.`
`AcmeTagSelectEvent.js`
Extends:
`oracle.adf.view.js.component.`
`AdfComponentEvent.js`
	Event:
	`com.adfdemo.acme.faces.event.`
	`TagSelectEvent.java`
	Extends:
	`javax.faces.event.FacesEvent.java`

	Event Listener:
	`com.adfdemo.acme.faces.event.`
	`SelectListener`
	Extends:
	`com.faces.event.FacesListener`

Component Peer:
`com.adfdemo.acme.js.component.`
`AcmeTagPanePeer.js`
Extends:
`oracle.adf.view.js.laf.rich.`
`AdfRichUIPeer.js`	

	Component Renderer:
	`com.adfdemo.acme.faces.render.`
	`TagPaneRenderer.java`
	Extends:
	`oracle.adf.view.rich.render.RichRenderer.java`

	Component JSP Tag:
	`oracle.adfdemo.acme.faces.taglib.`
	`TagPaneTag.java`
	Extends:
	`javax.faces.webapp.UIComponentELTag.java`

Table 34–2 (Cont.) Client-Side and Server-Side Artifacts for the tagPane Custom Component

Client	Server

	Composite Resource Loader:
	`oracle.adfdemo.acme.faces.resource.`
	`AcmeResourceLoader.java`
	Extends:
	`org.myfaces.trinidad.resource.`
	`RegxResourceLoader.java`

	JavaScript Resource Loader:
	`oracle.adfdemo.acme.faces.resource.`
	`ScriptsResourceLoader.java`
	Extends:
	`org.myfaces.trinidad.resource.`
	`AggregateingResourceLoader.java`

	Resource Bundle:
	`oracle.adfdemo.acme.faces.resource.`
	`AcmeSimpleDesktopBundle.java`
	Extends:
	`java.util.ListResouceBundle.java`

For code examples for some of these artifacts, see Section F.4, "Samples for Chapter 34, "Creating Custom ADF Faces Components"."

34.2 Setting Up the Workspace and Starter Files

Use JDeveloper to set up an application and a project to develop the custom component. After your skeleton project is created, you can add a deployment profile for packaging the component into a JAR file.

During the early stages of development, you create starter configuration and support files to enable development. You may add to and edit these files during the process. You create the following configuration files:

- `META-INF/faces-config.xml`: The configuration file required for any JSF-based application. While the component will use the `faces-config.xml` file in the application into which it is eventually imported, you will need this configuration file for development purposes.
- `META-INF/trinidad-skins.xml`: The configuration information for the skins that the component can use. Extend the simple skin provided by ADF Faces to include the new component.
- `META-INF/`*`package_directory`*`/styles/`*`skinName.css`*: The style metadata needed to skin the component.
- `META-INF/servlets/resources/`*`name`*`.resources`: The render kit resource loader that loads style sheets and images from the component JAR file. The resource loader is aggregated by a resource servlet in the web application, and is used to configure the resource servlet. In order for the servlet to locate the resource loader file, it must be placed in the `META-INF/servlets/resources` directory.

- `META-INF/adf-js-features.xml`: The configuration file used to define a feature. The definition usually includes a component name or description of functionality that a component provides, and the files used to implement the client-side component.
- `META-INF/`*`prefix_name`*`.tld` (for JSP): The tag definition library for the component. If the consuming web application is using JSP, the custom component requires a defined TLD. The TLD file will be located in the `META-INF` directory along with the `faces-config.xml` and `trinidad-skins.xml` files.
- `META-INF/`*`prefix_name`*`.taglib.xml` (for Facelets): The tag library definition for the component when the consuming application uses Facelets. This file defines the handler for the component.

For example, for the `tagPane` component, the following configuration files are needed:

- `META-INF/faces-config.xml`
- `META-INF/trinidad-skins.xml`
- `META-INF/acme/styles/acme-simple-desktop.css`
- `META-INF/servlets/resources/acme.resources`
- `META-INF/acme.tld`
- `META-INF/acme.taglib.xml`
- `META-INF/adf-js-features.xml`

After the files are set up in JDeveloper, you add content to them. Then, you create the client-side files nd server-side files. For more information, see Section 34.3, "Developing for the Client-Side," and Section 34.4, "Developing for the Server-Side."

34.2.1 How to Set Up the JDeveloper Custom Component Environment

This chapter assumes you have experience using JDeveloper and are familiar with the steps involved in creating and deploying an application. For more information about using JDeveloper to create applications, see Chapter 3, "Getting Started with ADF Faces and JDeveloper." For more information about deployment, see the "Deploying Fusion Web Applications" chapter of *Developing Fusion Web Applications with Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To set up the custom component development environment in JDeveloper:

1. Create an application to serve as a development container for the component. Use JDeveloper to create a workspace and project. For procedures on creating an application, see Section 3.2, "Creating an Application Workspace." When selecting an application template, select the Generic Application template.

Note: Do not select any other application template, or add any technologies to your application. Because the custom component will be packaged into a JAR file, you do not need to create unnecessary folders such as `public_html` that JDeveloper creates by default when you use a template specifically for web applications, or add web technologies. Instead, create the starter configuration file from the XML schemas.

---

2. Prepare the project to be deployed as a JAR file by creating a new deployment profile.
   a. In the Applications window, right-click the project and choose **New > From Gallery**.
   b. In the New Gallery, select **Deployment Profiles** and then **ADF Library JAR File**, and click **OK**.
   c. In the Create Deployment Profile dialog, enter a name for the Deployment Profile name. For example, the `tagPane` component might use `adf-richclient-demo-acme`.
   d. In the Edit JAR Deployment Profile Properties dialog, click **OK**.
3. In the Project Properties dialog, add library dependencies.
   a. Select **Libraries and Classpath** in the left pane.
   b. Click **Add Library**.
   c. In the Add Library dialog, select **ADF Faces Runtime 11**, **Facelets Runtime** (if using Facelets), **JSF 2.1**, and **JRF Runtime**, and click **OK**.
   d. Click **OK** to close the Project Properties dialog.
4. Register XML schemas.

   The custom component requires several XML configuration files. You can use JDeveloper to register the XML schemas associated with these configuration files. You must add schemas for three configuration files: `faces-config.xml`, `trinidad-skins.xml`, and `trinidad-config.xml`. By preregistering these schemas, you can create a template XML configuration file without having to know the specifics about the markup structure. The names and locations of the schemas are assumed by the base installation of JDeveloper.

   a. Select **Tools > Preferences**. In the Preferences dialog, select **XML Schemas** in the left pane, and click **Add**.
   b. In the Add Schema dialog, click **Browse** to navigate to the XML schemas included in your JDeveloper build, as shown in Table 34–3.

Note: In the Add Schema dialog, make sure **Extension** is set to `.xml`. If you change it to XSD, when you later create XML files, you will not be able to use the XML schema you have created.

Table 34–3 XML Schema Locations

XML Configuration File	Schema Location
/META-INF/faces-config.xml	*JDeveloper_Home*/jdeveloper/modules/ oracle.jsf_1.2.9/glassfish.jsf_1.2.9jar! /com/sun/faces/web-facesconfig_1_2.xsd
/META-INF/trinidad-skins.xml	*JDeveloper_Home*/jdeveloper/modules/oralce.adf.view_11.1.1/trinidad-impl.jar!/org/apache/ myfaces/trinidadinternal/ui/laf/xml/schemas/skin/trinidad-skins.xsd
/META-INF/trinidad-config.xml	*JDeveloper_Home*/jdeveloper/modules/ oracle.adf.view_11.1.1/ trinidad-api.jar!/trinidad-config.xsd
/META-INF/adf-js-features.xml	JDeveloper_Home/jdeveloper/modules/ oracle.adf.view_11.1.1/ adf-richclient-api-11.jar! /adf-js-features.xsd

34.2.2 How to Add a Faces Configuration File

Although the custom component will be registered in the consuming application's `faces-config.xml` file, during development, the workspace requires a `faces-config.xml` file.

Note: Do not use any of JDeveloper's declarative wizards or dialogs to create the `faces-config.xml` file. These declarative methods assume you are creating a web application, and will add uneccessary artifacts to your custom component application.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create a faces-config.xml file for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **XML** and then **XML Document from XML Schema**, and click **OK**.
3. In the Create XML from XML Schema dialog:
   - **XML File**: Enter `faces-config.xml`.
   - **Directory**: Append `\src\META-INF` to the end of the directory entry.
   - Select **Use Registered Schemas** and click **Next**.
4. Enter the following:
   - **Target Namespace**: Select `http://java.sun.com/xml/ns/javaee`.
   - **Root Element**: Select `faces-config`.

Leave the defaults for the other fields, and click **Finish**.

The new file will automatically open in the XML editor.

5. Add the following schema information after the first line in the file:

```
<?xml version="1.0" encoding="US-ASCII"?>
 <faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee">
```

Adding a schema provides better WYSIWYG tool support.

34.2.3 How to Add a MyFaces Trinidad Skins Configuration File

Add a MyFaces Trinidad skins file to register the component's CSS file, which is used to define the component's styles.

To create a trinidad-skins.xml file for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General,** select **XML** and then **XML Document from XML Schema**, and click **OK**.
3. In the Create XML from XML Schema dialog:
   - **XML File**: Enter `trinidad-skins.xml`.
   - **Directory**: Append `\src\META-INF` to the end of the Directory entry.
   - Select **Use Registered Schemas**, and click **Next**.
4. Enter the following:
   - **Target Namespace**: Select `http://myfaces.apache.org/trinidad/skin`.
   - **Root Element**: Select `skins`.
   - Click **Finish**. The new file will automatically open in the XML editor.

34.2.4 How to Add a Cascading Style Sheet

Add a cascading style sheet to define component's style.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create a cascading style sheet for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, click **All Items,** select **File**, and click **OK**.
3. In the Create File dialog:
   - Enter a file name, for example, `acme-simple-desktop.css`.
   - Append `\src\META-INF\`*`component_prefix`*`\styles` to the end of the Directory entry, where *`component_prefix`* is the prefix that will be used in the component library. For example, for the `tagPane` component, `acme` is the prefix, therefore, the string to append would be `\META-INF\acme\styles`.

34.2.5 How to Add a Resource Kit Loader

Create an empty file and add the fully qualified class path to the custom resource loader.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create a resource loader for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, click **All Items**, select **File**, and click **OK**.
3. In the Create File dialog:
   - Enter *`component_prefix`*`.resources` for **File Name**, where *`component_prefix`* will be the prefix used in the component library. For example, for the `tagPane` component, `acme` is the prefix, therefore, the string to enter is `acme.resources`.
   - Append `\src\META-INF\sevlets\resources\` to the end of the Directory entry.

34.2.6 How to Add a JavaServer Pages Tag Library Descriptor File

You need a JSP TLD file to work with JSF pages.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create a JavaServer Pages TLD file for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **Web Tier**, select **JSP** and then **JSP Tag Library**, and click **OK**.
3. In the Create JavaServer Page Tag Library dialog, select **Deployable** and click **Next**.
4. Enter the following:
   - **Tag Library Descriptor Version**: Select **2.1**.
   - **Short Name (Used as filename)**: A name. For example, for the **tagPane** component, you would enter `acme`.
   - **Tag Library Version**: Version of the tag Library. For example, 1.0.
   - **Tag Library URI**: A URI for the tag library. For example, for the **tagPane** component, you would enter `http://oracle.adfdemo.acme`.
   - **Tag Library Information**: Enter more information or description about the Tag Library.
5. Click **Next** and optionally enter additional tag library information, then click **Finish**.

34.2.7 How to Add a JavaScript Library Feature Configuration File

Add a features file to define the JavaScript files associated with the custom component, including the files for the client component, the client peer, and the client events.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create an adf-js-features.xml file for the custom component:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General,** select **XML** and then **XML Document from XML Schema**, and click **OK**.
3. In the Create XML from XML Schema dialog:
   - **XML File**: Enter `adf-js-features.xml`.
   - **Directory**: Append `\src\META-INF` to the end of the Directory entry.
   - Select **Use Registered Schemas**, and click **Next**.
4. Do the following:
   - **Target Namespace**: Select `http://xmlns.oracle.com/adf/faces/feature`.
   - **Root Element**: Select `features`.
   - Click **Finish**. The new file will automatically open in the XML editor.

34.2.8 How to Add a Facelets Tag Library Configuration File

If a consuming application uses Facelets, then you must define the handler for the component.

Before you begin:

It may be helpful to have an understanding of the workspace and starter files needed for custom ADF Faces components. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create a Facelets tag library file:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General,** select **XML** and then **XML Document,** and click **OK**.
3. In the Create XML file dialog, enter the following:
   - **File Name**: Enter *`prefix_name`*`.taglib.xml`
   - **Directory**: Append `\src\META-INF` to the end of the Directory entry.
4. Copy and paste the code shown in Example 34–4:

Example 34–4 Code for Facelets Tag Library Configuration File

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<!DOCTYPE facelet-taglib
          PUBLIC "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib xmlns="http://java.sun.com/JSF/Facelet">

  <namespace>http://xmlns.oracle.adfdemo/acme</namespace>

    <tag>
      <tag-name>tagPane</tag-name>
      <handler-class>
        oracle.adfinternal.view.faces.facelets.rich.RichComponentHandler
      </handler-class>
    </tag>
</facelet-taglib>
```

5. Replace the namespace and tag-name code shown in bold with code appropriate for your application.

34.3 Developing for the Client-Side

After the JDeveloper workspace and configuration files have been created, you can create and code the client-side JavaScript files. When you have finished with the client-side development, create the server-side files, as described in Section 34.4, "Developing for the Server-Side."

> **Best Practice:** Because JavaScript libraries do not have namespaces, you should create all JavaScript object names for the custom component using the same prefix. You do not need to do this on the server because the server-side Java package names will prevent name collisions. For example, for the `tagPane` component, the client-side JavaScript object names all have the `acme` prefix.

Client components hold state for properties that are not defined within the corresponding DOM element. These properties are bound to an associated DOM element using the `clientId`. The `clientId` uniquely defines a server-side component within the component tree representing a page. The DOM element holds the `clientId` within the `Id` attribute.

> **Note:** Place each JavaScript object in its own separate source file for best practice and consistency.

Developing the client-side component requires creating a JavaScript file for the component, the peer, and the component event.

In addition to the client component, client-side events must be defined. The `tagPane` component's client-side event is fired and propagated to the server when the user clicks one of the three file types. The client event passed to the server is queued so that the target server-side component can take the appropriate action.

Finally, the custom component requires a client peer. The peer is the component presenter. Peers act as the links between a client component and an associated DOM element. Client peers add client behaviors. A peer must be bound to a component through a registration method.

As with the client component, the associated peer is bound to a DOM element using the component's `clientId`. There are two types of peers, statefull and stateless.

- Some complex client components require the peer to hold state and thereby need to use a statefull peer. This type of peer is always bound to a DOM element. Statefull peers are less common than stateless peers.
- Stateless peers do not hold state and one peer can be bound to multiple components. Stateless peers are the best performance option because they reduce the client footprint. This type of peer performs lazy content delivery to the component.

Peers add behavior to the component by dynamically registering and listening for DOM events. Conceptually, a peer's function is similar to the role of a managed bean. However, the client component is not bound to the peer using EL like the server-side component is bound to a view model (`#{backingbean.callback}`). The peer registers client component events in the `InitSubclass` (`AdfRichUIPeer.addComponentEventHandlers("click")`) callback method. The callback is assumed by using a naming convention of (`<Peer>.prototype.HandleComponent<Event>`). The peer manages DOM event callbacks where the server-side component handles the linking using EL bindings to managed beans. For more information about client-side architecture, including peers, see Section 4.1, "About Using ADF Faces Architecture."

The following section assumes you have already set up a custom component development template environment. This development environment includes the setting up of application workspace, projects, deployment profiles, and registering schemas. If you have not done so, see Section 34.2, "Setting Up the Workspace and Starter Files."

34.3.1 How to Create a JavaScript File for a Component

Use JDeveloper to create a JavaScript file for the component. In it, you will define the component type for the component.

Before you begin:

It may be helpful to have an understanding of client-side development for custom ADF Faces components. For more information, see Section 34.3, "Developing for the Client-Side."

You must also have set up your workspace and created the configuration files. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create the component JavaScript file:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **Web Tier,** select **HTML** and then **JavaScript File**, and click **OK**.
3. In the Create JavaScript File dialog, do the following:
   - **File Name**: Enter the name of the client-side component. For example, for the `tagPane` component, you might enter `AcmeTagPane.js`.

     Tip: To prevent naming collisions, start the name with the component prefix.

   - **Directory**: Enter the directory path of the component in a subdirectory under the `src` directory. For example, for the `tagPane` component, you might enter `adfrichclient-demo-acme\src\oracle\adfdemo\acme\js\component`.

4. In the editor for the JavaScript file, add the component code to define the component type. Example 34–5 shows the code that might be used for the `tagPane` component.

Example 34–5 tagPane Component JavaScript

```
AdfUIComponents.createComponentClass(
 "AcmeTagPane",
 {
      componentType:"oracle.adfdemo.acme.TagPane",superclass:AdfUIObject
 }
);
```

34.3.2 How to Create a Javascript File for an Event

Use JDeveloper to create a JavaScript file for the event. Add code to the JavaScript to perform the functions required when a event is fired, such as a mouse click.

Before you begin:

It may be helpful to have an understanding of client-side development for custom ADF Faces components. For more information, see Section 34.3, "Developing for the Client-Side."

You must also have set up your workspace and created the configuration files. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create the JavaScript for the event:

1. In the Applications window, right-click the project and choose **New> From Gallery**.
2. In the New Gallery, expand **Web Tier**, select **HTML** and then **JavaScript File**, and click **OK**.
3. In the Create JavaScript File dialog, do the following:
   - **File Name**: Enter the name of the client-side event. For example, for the `tagPane` component, you might enter `AcmeTagSelectEvent.js`.

     Tip: To prevent naming collisions, start the name with the component prefix.

   - **Directory**: Enter the directory path of the event in a subdirectory under the `src` directory. For example, for the `tagPane` component, you might enter `adf-richclient-demo-acme\src\oracle\adfdemo\acme\js\event`.
4. In the editor for the JavaScript file, add the event code. For an example that shows the event code that might be added for the `tagPane` component, see Section F.4.1, "Event Code for JavaScript."

34.3.3 How to Create a JavaScript File for a Peer

Use JDeveloper to create a JavaScript file for the peer. Add code to register the peer and bind it to the component.

Before you begin:

It may be helpful to have an understanding of client-side development for custom ADF Faces components. For more information, see Section 34.3, "Developing for the Client-Side."

You should have also already created the JavaScript file for the component. For more information, see Section 34.3.1, "How to Create a JavaScript File for a Component."

To create the peer JavaScript file:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **Web Tier**, select **HTML** and then **JavaScript File**, and click **OK**.
3. In the Create JavaScript File dialog, do the following:
   - **File Name**: Enter the name of the client-side peer. For example, for the `tagPane` component, you might enter `AcmeTagPanePeer.js`.

     Tip: To prevent naming collisions, start the name with the component prefix.

   - **Directory**: Enter the directory path of the event in a subdirectory under the `src` directory. For example, for the `tagPane` component, you might enter `adf-richclient-demo-acme\src\oracle\adfdemo\acme\js\component`.
4. In the editor for the JavaScript file, add code for the peer. In this code, you must create the peer, add event handling with respect to the DOM, and register the peer with the component. Example 34–6 shows the code that might be added for the `tagPane` component.

Example 34–6 tagPane JavaScript Peer

```
AdfRichUIPeer.createPeerClass(AdfRichUIPeer, "AcmeTagPanePeer", true);
AcmeTagPanePeer.InitSubclass = function()
{
AdfLogger.LOGGER.logMessage(AdfLogger.FINEST,
 "AcmeTagPanePeer.InitSubclass()");
  AdfRichUIPeer.addComponentEventHandlers(this,
     AdfUIInputEvent.CLICK_EVENT_TYPE);
}

AcmeTagPanePeer.prototype.HandleComponentClick = function(componentEvent)
{
AdfLogger.LOGGER.logMessage(AdfLogger.FINEST,
"AcmeTagPanePeer.HandleComponentClick(componentEvent)");
  // if the left mouse button was pressed
if (componentEvent.isLeftButtonPressed())
 {
   // find component for the peer
  var component = this.getComponent();
  AdfAssert.assertPrototype(component, AcmeTagPane);
   // find the native dom element for the click event
var target = componentEvent.getNativeEventTarget();
    if (target && target.tagName == "A")
   {
   AdfLogger.LOGGER.logMessage(AdfLogger.FINEST, "File type element (A)
 found: " + componentEvent.toString());
     var tag = target.firstChild.nodeValue;
```

```
   AdfAssert.assertString(tag);


   AdfLogger.LOGGER.logMessage(AdfLogger.FINEST, "tag :" + tag);
   // fire a select event
AcmeTagSelectEvent.queue(component, tag);
   //cancel the native dom onclick to prevent browser actions based on the
   //'#' hyperlink. The event is of type AdfIEUIInputEvent. This event
   //will cancle the native dom event by calling
    //AdfAgent.AGENT.preventDefault(Event)
      componentEvent.cancel();
  }
// event has dom node
  }
}
// Register the peer with the component. This bit of script must
// be invoked after the AcmeTagPane and AcmeTagSelectEvent objects
// are created. This is enforced by the ordering of the script files
// in the
 oracle.asfdemo.acme.faces.resource.AcmeResourceLoader.
 AcmeScriptsResourceLoader.AdfPage.PAGE.getLookAndFeel()
.registerPeerConstructor("oracle.adfdemo.acme.TagPane",
        "AcmeTagPanePeer");
```

34.3.4 How to Add a Custom Component to a JavaScript Library Feature Configuration File

Now that you have created all the JavaScript files for the component, you can add the component to the `adf-js-features.xml` file you created.

Before you begin:

It may be helpful to have an understanding of client-side development for custom ADF Faces components. For more information, see Section 34.3, "Developing for the Client-Side."

You should have also already created the JavaScript file for the component. For more information, see Section 34.3.1, "How to Create a JavaScript File for a Component."

To add a custom component to a JavaScript library feature configuration file:

Follow the procedures documented in Section 4.9.1, "How to Create a JavaScript Feature," omitting the steps for creating the XML files, as you have already done so. Example 34–7 shows the `adf-js-features.xml` file used for the `tagPane` component.

Example 34–7 adf-js-features.xml File for the tagPane Component

```
<?xml version="1.0" encoding="UTF-8" ?>
<features xmlns="http://xmlns.oracle.com/adf/faces/feature">
  <feature>
    <feature-name>AcmeTagPane</feature-name>
    <feature-class>
      oracle/adfdemo/acme/js/component/AcmeTagPane.js
    </feature-class>
    <feature-class>
      oracle/adfdemo/acme/js/event/AcmeTagSelectEvent.js
    </feature-class>
    <feature-class>
      oracle/adfdemo/acme/js/component/AcmeTagPanePeer.js
    </feature-class>
```

```
  </feature>
</features>
```

34.4 Developing for the Server-Side

Server-side development involves creating Java classes for:

- Event listener: This class listens for events and then invokes processing logic to handle the event.
- Events: You create an event in order to invoke the logic in the associated listener.
- Component: This class holds the properties that define behavior for the component.
- Resource bundle: This class holds text strings for the component.
- Renderer: This class determines how the component will be displayed in the client.
- Resource loader: This class is required only if your component contains images needed for skinning.

After you have created the classes, add the component class and the renderer class to the `faces-config.xml` file. Then, complete the configuration files started in Section 34.2, "Setting Up the Workspace and Starter Files."

34.4.1 How to Create a Class for an Event Listener

The ADF Faces event API requires an event listener interface to process the event. The custom component has a dependency with the event and the event has a dependency with an event listener interface. The Java import statements must reflect these dependencies. You also must define the `componentType` for the component.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must also have set up your workspace and created the configuration files. For more information, see Section 34.2, "Setting Up the Workspace and Starter Files."

To create the EventListener class:

1. In the Applications window, right-click the project and choose **New> From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Interface,** and click **OK**.
3. In the Create Java Interface File dialog, do the following:
   - **Name**: Enter a listener name. For example, for the `tagPane` component, you might enter `TagSelectListener`.
   - **Package**: Enter a name for the package. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.event`.
4. In the editor for the Java file, add the following:
   - Have the listener extend the `javax.faces.event.FacesListener` interface.

- Add an `import` statement, and import the `FacesListener` class and any other classes on which your event is dependent.
- Add a method signature that will process the new event. Even though you have not created the actual event, you can enter it now so that you will not have to enter it later.

Example 34–8 shows the code for the `tagPane` event listener.

Example 34–8 tagPane Event Listener Java Code

```
package oracle.adfdemo.acme.faces.event;

import javax.faces.event.AbortProcessingException;
import javax.faces.event.FacesListener;

public interface TagSelectListener
 extends FacesListener
{
 /**
  * <p>Process the {@link TagSelectEvent}.</p>
  * @param event fired on click of a tag link
  * @throws AbortProcessingException error processing {@link TagSelectEvent}
  */
 public void processTagSelect(TagSelectEvent event)
  throws AbortProcessingException;
}
```

34.4.2 How to Create a Class for an Event

You must create a server-side event that will be the counter representation of the JavaScript event created in Section 34.3.2, "How to Create a Javascript File for an Event." Server-side JSF events are queued by the component during the Apply Request Values lifecycle phase. Events propagate up to the `UIViewRoot` class after all the phases but the Render Response phase. Queued events are broadcast to the associated component.

The server-side Java component must raise the server-side event, so you must create the event source file first to resolve the compilation dependency.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You should have also already created the event listener class. For more information, see Section 34.4.1, "How to Create a Class for an Event Listener."

To create the server-side event class:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General,** select **Java** and then **Java Class,** and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter an event name. For example, for the `tagPane` component, you might enter `TagSelectEvent`.

- **Package**: Enter the package name. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.event`.
- **Extends**: Enter a name for the class that the event class extends. This is usually `javax.faces.event.FacesEvent`.
- In the Optional Attributes section, select the following:.
  - In the **Access Modifiers** section, select **public**.
  - At the bottom, select **Constructors from Superclass** and **Implement Abstract Methods**.

Example 34–9 shows the code for the event class.

Example 34–9 tagPane Event Java Code

```
package oracle.adfdemo.acme.faces.event;
import javax.faces.component.UIComponent;
import javax.faces.event.FacesEvent;
import javax.faces.event.FacesListener;

public class TagSelectEvent
 extends FacesEvent
{
 /**
  * <p>Tag selected on the client.</p>
  */
 private String tag = null;
 /**
  * <p>Overloade constructor passing the <code>source</code>
  * {@link oracle.adfdemo.acme.faces.component.TagPane} component and the
  * selected <code>tag</code>.
  * </p>
  * @param source component firing the event
  * @param tag selected tag link type
  */
 public TagSelectEvent(UIComponent source,
            String tag)
 {
  super(source);
  this.tag = tag;
 }

 /**
  * <p>Returns <code>true</code> if the <code>facesListener</code> is a
  * {@link TagSelectListener}.</p>
  *
  * @param facesListener listener to be evaluated
  * @return <code>true</code>
  * if <code>facesListener</code> instancof {@link TagSelectListener}
  */
 public boolean isAppropriateListener(FacesListener facesListener)
 {
  return (facesListener instanceof TagSelectListener);
 }
 /**
  * <p>Delegates to the <code>processTagSelect</code>
  * method of a <code>FacesListener</code>
  * implementing the {@link TagSelectListener} interface.
  *
  * @param facesListener target listener realizing {@link TagSelectListener}
```

```
 */
public void processListener(FacesListener facesListener)
{
 ((TagSelectListener) facesListener).processTagSelect(this);
}
/**
 * @return the tag that was selected triggering this event
 */
public String getTag()
{
 return tag;
}
}
```

34.4.3 Creating the Component

A JSF component can be described as a state holder of properties. These properties define behavior for rendering and how a component responds to user interface actions. When you are developing the component class, you identify the types of the needed properties. You also define the base component that it will extend from the MyFaces Trinidad Framework. For example, the `tagPane` component extends the `UIXObject` in MyFaces Trinidad.

Most components will have several properties that should be implemented. Some of the properties are inherited from the base class, and some are required for the ADF Faces framework. Other properties are required because they are best practice. And finally, some properties are specific to the functionality of the custom component.

For example, the `tagPane` component has the properties shown in Table 34–4.

Table 34–4 Component Properties for the tagPane Custom Component

Origin	Property	Data Type	Description
Inherited	`id`	`String.class`	The identifier for a component.
	`rendererType`	`String.class`	The logical identifier registered as a component renderer.
	`rendered`	`Boolean.class`	True or false flag that determines if the component is rendered.
	`binding`	`ValueExpression.class`	A binding value expression to store a component instance in a managed bean.
ADF Faces Framework	`clientComponent`	`Boolean.class`	True or false flag that determines whether a client-side component will be generated.
	`clientListeners`	`ClientListenerSet.class`	A binding expression that registers a client listener on a component.
	`clientAttributes`	`Set.class`	A client attribute on a component. The attribute is added both to the server-side JSF component as well as the client-side equivalent.
Best Practice	`inlineStyle`	`String.class`	A CSS style applied to the root component's class attribute.
	`styleClass`	`String.class`	A CSS style added to the component's class attribute.
	`visible`	`Boolean.class`	True or false flag that returns the visibility of the component. The visible property is not the same as the rendered property. The visible attribute affects the CSS style on the CSS root of the component.
	`partialTriggers`	`String[].class`	The IDs of the components that should trigger a partial page update.

Table 34–4 (Cont.) Component Properties for the tagPane Custom Component

Origin	Property	Data Type	Description
Specific to tagPane	`tags`	`Map.class`	The map of weighted tags. The key represents the tag name and the value as a number. `Map<String.Number>`.
	`orderBy`	`String.class`	The order that the tags are rendered. The valid enumerations are `alpha` and `weight`.
	`tagSelectListener`	`MethodExpression.class`	The `newselectListener` method binding expression that expects a single parameter of type `oracle.adfdemo.acme.faces.event.TagSelectEvent`. This binding will be when the client-side `oracle.adfdemo.acme.js.event.AcmeTagSelectEvent.js` event is queued from clicking one of the tags.

ADF Faces and MyFaces Trinidad component libraries are defined differently from other libraries. A JSF component has a collection called `attributes` that provides access to component properties (using the Java simple beans specification) through a `MAP` interface. The collection also holds value pairs that do not correspond to a component's properties. This concept is called *attribute transparency*. The JSF runtimes (both MyFaces Trinidad and the JSF reference implementation) implement this concept using the Java reflection API.

My Faces Trinidad defines its own internal collection, which does not use the Java reflection API. This difference means that it is more efficient than the base implementation. The solution in MyFaces Trinidad collects more metadata about the component properties. This metadata declares state properties, which allows the base class to fully implement the `StateHolder` interface in a base class.

My Faces Trinidad extends the `javax.faces.component.UIComponent` class with the `org.apache.trinidad.component.UIXComponent` class, followed by a complete component hierarchy. To ease code maintenance, the framework has a strategy for generating code based on configuration files and templates.

This component strategy is a trade-off in terms of development. It requires more coding for defining properties, but you will not have to code the two methods (`saveState`, `restoreState`) for the `StateHolder` interface for each component.

Note: Do not have your custom component extend from any ADF Faces implementation packages. These implementations are private and might change.

34.4.4 How to Create a Class for a Component

Use JDeveloper to create a Java file for the component. Create a `Type` bean to hold property information and define a `PropertyKey` for each property. Then, generate accessors for the private attributes.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side." You should also understand the class you will be creating. For more information, see Section 34.4.3, "Creating the Component."

You must have also created the event class. For more information, see Section 34.4.2, "How to Create a Class for an Event."

To create the component class:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class**, and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter a component name. For example, for the `tagPane` component, you might enter `TagPane`.
   - **Package**: Enter a name for the package. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.component`.
   - **Extends**: Enter a name for the class the component class extends. For example, for the `tagPane` component, you would enter `org.apache.myfaces.trinidad.component.UIXObject`.
   - In the Optional Attributes section, select the following:.
     – In the **Access Modifiers** section, select **public**.
     – At the bottom, select **Constructors from Superclass,** and **Implement Abstract Methods**.
4. In the source editor, create a `Type` bean that contains component property information. This static class attribute shadows an attribute with the same name in the superclass. The `type` attribute is defined once per component class. Through the `Type` constructor, you pass a reference to the superclass's `Type` bean, which copies property information. For example, the `tagPane` class would contain the following constructor:

   ```
   static public final FacesBean.Type TYPE = new FacesBean.Type(UIXObject.TYPE);
   ```

5. For each property, define a static `PropertyKey` that is used to access the properties state. Use the `TYPE` reference to register a new attribute. Specify the property type using the class reference. The component data type should correspond to the component property. There is another overload of the `registerKey` method that allows you to specify state information. The default assumes the property is `persistent`. Example 34–10 shows the `PropertyKey` methods for the `tagPane` component.

Example 34–10 PropertyKey Definition

```
/**
 * <p>Custom CSS applied to the style attribute of the root markup node.</p>
 */
static public final PropertyKey INLINE_STYLE_KEY =
 TYPE.registerKey("inlineStyle", String.class);
/**
 * <p>Custom CSS class to the class attribute of the root markup node.</p>
 */
static public final PropertyKey STYLE_CLASS_KEY =
 TYPE.registerKey("styleClass", String.class);
```

6. Right-click in the editor and choose **Generate Accessors**. In the Generate Accessors dialog, click **Select All**, ensure the Scope is set to **Public**, and click **OK**. This allows JDeveloper to generate `get` and `set` methods for the private attributes.

   Then, remove the private attribute and replace with calls to `getProperty(PropertyKey)` and `getProperty(PropertyKey)`.

Example 34–11 shows the code after replacing the private attribute.

Example 34–11 Component Properties

```
public void setInlineStyle(String newinlineStyle)
{
 // inlineStyle = newinlineStyle;
 setProperty(INLINE_STYLE_KEY, newinlineStyle);
}
/**
 * <p>CSS value applied to the root component's style attribute.</p>
 *
 * @return newinlineStyle CSS custom style text
 */
public String getInlineStyle()
{
 // return inlineStyle;
 return (String) getProperty(INLINE_STYLE_KEY);
}
```

7. You may need to override any methods to perform specific functions in the component. For example, to allow your component to participate in partial page rendering (PPR), you must override the `getBeanType` method, as shown in Example 34–12.

Example 34–12 Overridden Methods

```
/**
 * <p>Exposes the <code>FacesBean.Type</code> for this class through a protected
 * method.  This method is called but the <code>UIComponentBase</code> superclass
 * to setup the components <code>ValueMap</code> which is the container for the
 * <code>attributes</code> collection.</p>
 *
 * @return <code>TagPane.TYPE</code> static property
 */
@Override
protected FacesBean.Type getBeanType()
 {
   return TYPE;
 }
```

Refer to the *Java API Reference for Oracle ADF Faces* for more information about the class your component extends, and the methods you may need to override.

For the `tagPane` component, the component must act on the event fired from the client component. A reference to the source component is passed as a parameter to the event's constructor.

For the `tagPane` component, the `broadcast` method checks if the event passed in using the formal parameter is a `TagSelectEvent`. If it is, the `broadcast` method invokes the method expression held by the `tagSelectListener` attribute.

Most events have an `immediate` boolean property that specifies the lifecycle phase in which the event should be invoked. If the `immediate` attribute is `true`, the event is processed in the Apply Values phase; otherwise, the event is processed in the Invoke Application phase. For more information, see Chapter 5, "Using the JSF Lifecycle with ADF Faces."

Example 34–13 shows the overwritten `broadcast` method for the `tagPane` component.

Example 34–13 The broadcast Method in the tagPane Component

```
/**
 * @param facesEvent faces event
 * @throws AbortProcessingException exception during processing
 */
@Override
public void broadcast(FacesEvent facesEvent)
 throws AbortProcessingException
{
 // notify the bound TagSelectListener
 if (facesEvent instanceof TagSelectEvent)
 {
  TagSelectEvent event = (TagSelectEvent) facesEvent;
  // utility method found in UIXComponentBase for invoking method event
  // expressions
  broadcastToMethodExpression(event, getTagSelectListener());
 }
 super.broadcast(facesEvent);
}
```

For a complete example of the Java class, see Section F.4.3, "Example Component Class Code."

34.4.5 How to Add the Component to the faces-config.xml File

After creating the component class, register the component by adding it to the `/META-INF/faces-config.xml` file. By defining the component in the faces configuration file packaged with the JAR project, you ensure that component is automatically recognized by the JSF runtime during web application startup.

To register the component, enter the component type, which is a logical name used by the applications factory to instantiate an instance of the component. For example, the `tagPane` component's type is `oracle.adfdemo.acme.TagPane`. You also need to add the fully qualified class path for the component, for example `oracle.adfdemo.acme.faces.component.TagPane`.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side." You should also understand the class you will be creating. For more information, see Section 34.4.3, "Creating the Component."

You must have also created the component class. For more information, see Section 34.4.4, "How to Create a Class for a Component."

To register a custom component:

1. In the Applications window, double-click the **faces-config.xml** file.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Components** navigation tab and click the **Add** icon.
4. Enter the type and class for the component.
5. Optionally, add any attributes, properties, or facets.

Example 34–14 shows the `tagPane` component defined within a `faces-config.xml` file.

Example 34–14 tagPane Component Added to the faces-config.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee">
 <application>
 </application>

 <component>
  <component-type>oracle.adfdemo.acme.TagPane</component-type>
  <component-class>oracle.adfdemo.acme.faces.component.TagPane
  </component-class>
 </component>
```

34.4.6 How to Create a Class for a Resource Bundle

Resource bundles are used to store information for the component, such as text for labels and messages, as well as translated text used if the application allows locale switching. Skins also use resource bundles to hold text for components. Because your custom component must use at least the simple skin, you must create at least a resource bundle for that skin. For a custom component, create a Java file for the resource bundle. For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

> **Tip:** You can also use a properties file for your resources.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side." You should also understand the class you will be creating. For more information, see Section 34.4.3, "Creating the Component."

You must have also registered the component. For more information, see Section 34.4.5, "How to Add the Component to the faces-config.xml File."

To create the resource bundle class:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class,** and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter a resource bundle name. The name should reflect the skin with which it will be used. For example, for the sample component, you might enter `AcmeSimpleDesktopBundle`.
   - **Package**: Enter a name for the package. For example, for the sample component, you might enter `oracle.adfdemo.acme.faces.resource`.
   - **Extends**: For resource bundles, you *must* enter `java.util.ListResourceBundle`.
   - In the Optional Attributes section, select the following:.
     – In the **Access Modifiers** section, select **public**.
     – At the bottom, select **Constructors from Superclass** and **Implement Abstract Methods**.

4. Add any keys and define the text as needed. For more information about creating resource bundles for skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

   Example 34–15 shows the resource bundle code for the `tagPane` component.

Example 34–15 tagPane Resource Bundle Java Code

```
package oracle.adfdemo.acme.faces.resource;
import java.util.ListResourceBundle;
/**
 * <p>Holds properties used by the components bundled in the jar project.
 * This bundle is part of the trinidad component skin that is configured
 * in the "/META-INF/trinidad-skins.xml" file. Component Renderers
 * will use the <code>RenderingContext</code> to lookup a key by calling
 * the <code>getTranslatedString(key)</code> method.</p>
 */
public class AcmeSimpleDesktopBundle
 extends ListResourceBundle
{
 /**
  * <p>Returns a two dimensional object array that represents a resource bundle
. * The first
  * element of each pair is the key and the second the value.</p>
  *
  * @return an array of value pairs
  */
 protected Object[][] getContents()
 {
  return new Object[][]
   {
    {"AcmeTagPane_tag_title","Tag Weight: {0}"}
   };
 }
}
```

5. To register the resource bundle for the simple desktop skin and any other desired skins, double-click the **/META-INF/trinidad-skins.xml** file to open it and do the following:

   a. In the Structure window, select **skin-addition**.

   b. In the Properties window, enter a skin ID. For the simple skin ID, enter `simple.desktop`.

   c. In the Structure window, right-click **skin-addition** and choose **Insert inside skin-addition > bundle-name**.

   d. In the Properties window, enter the fully qualified name of the resource bundle just created.

   Note: JDeveloper adds `translation-source` and `bundle-name` elements as comments. Instead of declaratively creating another `bundle-name` element, you can manually enter the `bundle-name` value in the generated element, and then remove the comment tag.

   Example 34–16 shows the code for registering the `tagPane` resource bundle with the simple skin (you will add the `style-sheet-name` element value in a later step).

Example 34–16 Registering a Resource Bundle with a Skin

```
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
 <skin-addition>
  <skin-id>simple.desktop</skin-id>
  <style-sheet-name></style-sheet-name>
  <bundle-name>
    oracle.adfdemo.acme.faces.resource.AcmeSimpleDesktopBundle
  </bundle-name>
 </skin-addition>
</skins>
```

34.4.7 How to Create a Class for a Renderer

ADF Faces components delegate the functionality of the component to a component class, and when the consuming application uses JSPs, the display of the component to a renderer. By default, all tags for ADF Faces combine the associated component class with an HTML renderer, and are part of the HTML render kit. HTML render kits are included with ADF Faces for display on both desktop and PDA devices.

Renderers are qualified in a render kit by family and renderer type. The family is a general categorization for a component, and should be the same as the family defined in the superclass. You do not have to override the `getFamily()` method in the component because the component will have the method through inheritance.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side." You should also understand the class you will be creating. For more information, see Section 34.4.3, "Creating the Component."

You must have also created the resource bundle. For more information, see Section 34.4.6, "How to Create a Class for a Resource Bundle.":

To create the renderer class:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class,** and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter a renderer name. For example, for the `tagPane` component, you might enter `TagPaneRenderer`.
   - **Package**: Enter a name for the package. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.render`.
   - **Extends**: Enter `oracle.adf.view.rich.render.RichRenderer`.
   - In the Optional Attributes section, select the following:.
     – In the **Access Modifiers** section, select **public**.
     – At the bottom, select **Constructors from Superclass** and **Implement Abstract Methods**.
4. Add any needed functionality. For more information about the methods and fields available on the `RichRender` class, see the *Java API Reference for Oracle ADF Faces*.

Note: The skinning functionality provides an API you can use to get the CSS style properties for a given CSS selector during rendering of the component. This API is useful if you need to do conditional rendering based on what styling is set. For more information, see `RenderingContext#getStyles` and `Styles#getSelectorStyleMap` in the MyFaces Trinidad Javadoc at `http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html`.

Note: If your DOM structure is one that supports stretching, you can have your component automatically stretched to fill up the browser's viewport when that component is the sole visual root component in the component tree and if the document component is configured to allow stretching (for more information, see Section 9.2.5, "How to Configure the document Tag").

To have your component automatically stretch, override `RichRenderer.getPrependedInlineStyle()` and return `getVisualRootStretchingStyles(context, rc, component, client, bean).`

This method will return null in cases where your component is not the visual root component and will return a `String` of styles that would cause a DIV to fill up the browser's viewport if your component is the visual root component.

For a complete example of a renderer class, see Section F–10, "Renderer Class."

34.4.8 How to Add the Renderer to the faces-config.xml File

After you create the renderer, register it using the `faces-config.xml` configuration file. If you want the custom component to work with the other ADF Faces components, you must use the same render kit ID that ADF Faces components use.

Tip: The most granular level that JSF allows for defining a render kit is at the view root.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must have also created the render kit and renderer. For more information, see Section 34.4.7, "How to Create a Class for a Renderer."

To register the render kit and renderer:

1. In the Applications window, double-click the **faces-config.xml** file.
2. In the editor window, select the **Overview** tab and then select the **Render Kits** navigation tab.
3. Click the **Add** icon for the Render Kits and enter `oracle.adf.rich` for the render kit ID.

4. Register your renderer by clicking the **Add** icon for Renderers and doing the following:

   - **Family**: Enter the class that the component extends. For example, for the `tagPane` component, you would enter `org.apache.myfaces.trinidad.Object`.
   - **Type**: Enter the type for the component. For example, for the `tagPane` component, you would enter `oracle.adfdemo.acme.TagPane`. This *must* match the renderer type.
   - **Class**: Enter the fully qualified class path to the renderer created in Section 34.4.7, "How to Create a Class for a Renderer." For example, for the `tagPane` component, you would enter `oracle.adfdemo.acme.faces.render.TagPaneRenderer`.

Example 34–17 shows the registration of the `tagPane` component render kit and renderer.

Example 34–17 tagPane Renderer Added to the faces-config.xml File

```
<render-kit>
  <render-kit-id>oracle.adf.rich</render-kit-id>
  <renderer>
    <component-family>org.apache.myfaces.trinidad.Object</component-family>
    <renderer-type>oracle.adfdemo.acme.TagPane</renderer-type>
    <renderer-class>oracle.adfdemo.acme.faces.render.TagPaneRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

34.4.9 How to Create JSP Tag Properties

To use the component on a JSP page, you create a custom tag that will instantiate the custom component. The JSP tag has nothing to do with rendering because the component's renderer will actually perform that task. In JSF 1.1, the JSP tag would invoke rendering on the component after creating and adding it to the component tree. This caused problems because the non-JSF/JSP tags were writing to the same response writer. The timing of the interleaving did not work out for components that rendered their own child components.

> **Note:** An application that uses Facelets uses a handler to instantiate the component. For more information, see Section 34.2.8, "How to Add a Facelets Tag Library Configuration File."

In JSF 1.2, the target for Java EE 5 (Servlet 2.5, JSP 2.1), most of the JSP problems were fixed. The JSF/JSP component acts as a component factory that is responsible only for creating components. This means that the rendering response phase is divided into two steps. First the component tree is created, and then the tree is rendered, instead of rendering the components as the component tree was being built. This functionality was made possible by insisting that the entire view be represented by JSF components. The non-JSF/JSP generates markup that implicitly becomes a JSF verbatim component.

As a result of changing these mechanics, in JSF 1.2, custom JSP tags extend the `javax.faces.webapp.UIComponentELTag` class. The `encodeBegin`, `encodeChildren`, and `encodeEnd` methods in the JSP tag have been deprecated. These methods once made corresponding calls to the component. Because the view root in JSF 1.2 does the rendering, all the work can be done in the `doStartTag` and `doEndTag` methods. MyFaces Trinidad has its own version of this base class that you will use. The `org.apache.myfaces.Trinidad.webapp.UIComponentELTag` hooks into the components property bag and makes coding JSPs simpler.

The tag class includes the creation of the component's properties. You must choose tag properties carefully. There are some properties that you can ignore for tag implementation, but they may be required as TLD attributes.

The following three attributes are implemented by superclasses and shared by many components through Java inheritance:

- `id`
- `binding`
- `rendered`

Do not implement the `id` attribute because the `id` attribute is implemented by the superclass `javax.faces.webapp.UIComponentTagBase`. The superclass `javax.faces.webapp.UIComponentELTag` implements the other two attributes, `binding` and `rendered`. Therefore, you do not need to add these to your tag class.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must have also completed the previous procedures in this section, up to this point.

To add a JSP tag:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class,** and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter a tag name. For example, for the `tagPane` component, you might enter `TagPaneTag`.
   - **Package**: Enter a name for the package. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.taglib`.
   - **Class**: Enter `org.apache.myfaces.trinidad.webapp.UIXComponentELTag`.
   - In the Optional Attributes section, select the following:.
     – In the **Access Modifiers** section, select **public**.
     – At the bottom, select **Constructors from Superclass** and **Implement Abstract Methods**.
4. In the source editor, add all the attributes to the file.

   Example 34–18 shows the code for the attributes for the `TagPaneTag` class.

Example 34–18 Attributes in the TagPaneTag Class

```
public class TagPaneTag
 extends UIXComponentELTag
{
 private ValueExpression _partialTriggers = null;
```

```
private ValueExpression _visible = null;
private ValueExpression _inlineStyle = null;
private ValueExpression _styleClass = null;
private ValueExpression _tags = null;
private ValueExpression _orderBy = null;
private MethodExpression _tagSelectListener = null;
```

5. To declaratively generate the accessor methods for the attributes, right-click the file in the source editor and choose **Generate Accessors**.
6. In the Generate Accessors dialog, click **Select All**, set the **Scope** to `public` and click **OK**.
7. Add the render type and component type to the class. The component type will be used by the superclass to instantiate the component using the application's factory method, `createComponent(componentType)`.

   Example 34–19 shows the code for the `TagPaneTag` class, where both the component type and render type are `oracle.adfdemo.acme.TagPane`.

Example 34–19 Component Type and Render Type for the TagPaneTag Class

```
public String getComponentType()
{
 return COMPONENT_TYPE;
}
public String getRendererType()
{
 return RENDERER_TYPE;
}

/**
 * <p>This component's type, <code>oracle.adfdemo.acme.TagPane</code></p>
 */
static public final String COMPONENT_TYPE =
 "oracle.adfdemo.acme.TagPane";
/**
 * <p>Logical name given to the registered renderer for this component.</p>
 */
static public final String RENDERER_TYPE = "oracle.adfdemo.acme.TagPane";
```

8. Override the `setProperties` method from the superclass that has a single formal parameter of type `FacesBean`. This is a MyFaces Trinidad version on the base `UIComponentELTag`, but it is passed the components state holder instead of the component reference. The job of the `setProperties` method is to push the JSP tag attribute values to the component.

   Example 34–20 shows the overridden method for the `tagPaneTag` class.

Example 34–20 Overridden setProperties Method in the TagPaneTag Class

```
@Override
protected void setProperties(FacesBean facesBean) {
  super.setProperties(facesBean);

  setStringArrayProperty(facesBean, TagPane.PARTIAL_TRIGGERS_KEY,
              _partialTriggers);
  setProperty(facesBean, TagPane.VISIBLE_KEY, _visible);
  setProperty(facesBean, TagPane.INLINE_STYLE_KEY, _inlineStyle);
  setProperty(facesBean, TagPane.STYLE_CLASS_KEY, _styleClass);
  setProperty(facesBean, TagPane.TAGS_KEY, _tags);
```

```
        setProperty(facesBean, TagPane.ORDER_BY_KEY, _orderBy);
        facesBean.setProperty(TagPane.TAG_SELECT_LISTENER_KEY,
                   _tagSelectListener);

      }
```

34.4.10 How to Configure the Tag Library Descriptor

A tag library descriptor (TLD) provides more information on the Java Class to the JSP compilation engine and IDE tools (TLDs are not used in applications that use Facelets).

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must have also completed the previous procedures in this section, and have specifically associated the tag library with a URI, assigned a version, and given it a name. You should have already performed this step when you created the tag library stub file in Section 34.2.6, "How to Add a JavaServer Pages Tag Library Descriptor File."

To configure the TLD:

1. From the **File** menu choose **Open**. Navigate to the `/src/META-INF` directory of your application, and double click the TLD file you created to open it.
2. In XML page of the Components window, drag and drop a **tag** element.
3. In the Insert tag dialog, do the following:
   - **name**: Enter the name of the component. For example, for the `tagPane` component, you might enter `tagPane`.
   - **body-content**: Enter `JSP`.
   - **tag-class**: Click the ellipses button and navigate to the components tag class file.
4. Define each of the attributes as follows. For each attribute:
   a. In the Structure window, right-click the **tag** element and choose **Insert inside tag > attribute**.
   b. In the Insert Attribute dialog, enter a value for the name. This should be the same as the name given in the tag class.
   c. In the Structure window, select the attribute and in the Properties window, set any attribute values.

      There are three types of elements to define for each attribute. The `<id>` element is a simple string. Additionally attributes can be either deferred-value or deferred-method attributes. These allow late (deferred) evaluation of the expression. Now that JSP and JSF share the same EL engine, the compiled EL can be passed directly to the component.

   For an example of the TLD for the `tagPane` component, see Section F.4.2, "Example Tag Library Descriptor File Code."

34.4.11 How to Create a Resource Loader

A resource loader is required only if the custom component has image files needed for the component's skinning. The images files are packaged into the JAR project so that the consumer of the component library will need to include the JAR file into the class path of their web project and add a few entries into their web deployment descriptor file (`web.xml`). The ADF Faces framework uses a resource servlet to deliver images. You need to register this servlet in the `web.xml` file and then create the resource loader class. A component library requires a resource loader that is auto-loaded by the resource servlet. You create a URL pattern folder mapping for the servlet, which will be used to locate and identify resources within your custom component library.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must have also completed the previous procedures in this section, up to this point.

To create a resource loader class:

1. In the Applications window, right-click the project and choose **New > From Gallery**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class,** and click **OK**.
3. In the Create Java Class File dialog, do the following:
   - **Name**: Enter a resource loader name. For example, for the `tagPane` component, you might enter `AcmeResourceLoader`.
   - **Package**: Enter a name for the package. For example, for the `tagPane` component, you might enter `oracle.adfdemo.acme.faces.resources`.
   - **Extends**: Enter a name for the class that the tag extends. For example, for the `tagPane` component, you would enter `org.apache.myfaces.trinidad.resource.RegexResourceLoader`.
   - In the Optional Attributes section, select the following:.
     – In the **Access Modifiers** section, select **public**.
     – At the bottom, select **Constructors from Superclass** and **Implement Abstract Methods**.
4. In the source editor, register regular expressions that map to more specific resource loaders. For example, you might create an expression that maps image resources located under an images directory.

   Example 34–21 shows the expression for the `tagPane` component that maps the `/acme/images/` directory located relative to the `/META-INF` directory of the custom component JAR file. As a result of the registration, the custom component images should be located under `/META-INF/acme/images`.

Example 34–21 Resource Loader for the tagPane Component

```
public class AcmeResourceLoader
 extends RegexResourceLoader
{
 public AcmeResourceLoader()
 {
```

```
// any resource in "/acme/" with the following suffixes will be
// loaded from the base folder of "META-INF".
// The servlet pattern match "/acme/*" should exist under "META-INF".
// For example URL : context-root/acme/images/type1.gif
//      map to: META-INF/acme/images/type1.gif
register("(/.*\\.(jpg|gif|png|jpeg))",
     new ClassLoaderResourceLoader("META-INF"));
```

5. Register the Libraries Resource Loader by opening the `/META-INF/servlet/resources/`*`name`*`.resources` file and adding the fully qualified name of the resource loader class bound to the URI pattern.

   The MyFaces Trinidad `ResourceServlet` uses the servlet context to scan across all JAR files within the class path of the web application. The servlet looks at its own URI mappings in the web deployment descriptor to formulate the location of this resource file. This file must contain the fully qualified name of the Java class bound to the URI pattern. During startup, the `ResourceServlet` will locate and use this file in a manner similar to how `FacesServlet` locates and uses the `faces-config.xml` files.

   For the `tagPane` component, the `acme.resources` file would contain this entry for the composite resource loader:

   ```
   oracle.adfdemo.acme.faces.resource.AcmeResourceLoader
   ```

34.4.12 How to Create a MyFaces Trinidad Cascading Style Sheet

A skin is a style sheet based on the CSS 3.0 syntax specified in one place for an entire application. Instead of inserting a style sheet on each page, you use one or more skins for the entire application. Every component automatically uses the styles as described by the skin. No design time code changes are required.

All ADF Faces components use skins. The default skin is the Skyros skin. Because your custom components will be used in conjunction with other ADF Faces components, you add style selectors to an existing ADF Faces skin. Because the Skyros skin inherits styles from the simple skin, you can simply add your selectors to the simple skin, and it will be available in all skins. However, you may want to style the selector differently for each skin. You set these styles in the CSS file you created. This file will be merged with other CSS styles in the application in which the component is used.

The text used in a skin is defined in a resource bundle. Create the text by creating a custom resource bundle and declaring the text you want to display. After you create your custom resource bundle, you register it with the skin. Coupling resource bundles with your CSS provides a method to make your components support multiple locales.

The `/META-INF/trinidad-skins.xml` file you created is used to register your CSS file and your resource bundle with an ADF Faces skin.

Before you begin:

It may be helpful to have an understanding of server-side development for custom ADF Faces components. For more information, see Section 34.4, "Developing for the Server-Side."

You must have also completed the previous procedures in this section, up to this point.

To create styles for your component:

1. Open the CSS file you created in Section 34.2.4, "How to Add a Cascading Style Sheet."

2. Define a root style selector for the component. This style will be associated with the `<DIV>` element that establishes the component.
3. Add other style selectors as needed. Example 34–22 shows the CSS file for the `tagPane` component.

Example 34–22 CSS File for the tagPane component

```
acme|tagPane          -  root element
acme|tagPane::content -  container for the links
acme|tagPane::tag - tag hyperlink
```

For more information about creating CSS for components to be used by skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

4. Create any needed resource bundle for your component.
5. To register your CSS with an ADF Faces skin, open the `/META-INF/trinidad-skins.xml` file.
6. In the Structure window, select the **skin-addition** element, and in the Properties window, do the following:
   - **skin-id**: Enter the ADF Faces skin to which you want to add the custom component selectors. You must register the selectors at least to the `simple.desktop` skin in order for them to be compatible with ADF Faces components.

     Note: If there is a possibility that the component will be used in an Oracle WebCenter Portal application, then you must also register the selectors with the `simple.portlet` skin. Skins are also available for PDAs (for example, `simple.pda`). For more information, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

   - **style-sheet-name**: Use the dropdown menu to choose **Edit**, and navigate to the CSS file you created.
7. If you created a resource bundle, add the fully qualified path to the bundle as the value for the `<bundle-name>` element.

   Example 34–23 show the code for the `tagPane` component.

Example 34–23 tagPane trinidad-skins.xml Code

```
<?xml version="1.0" encoding="UTF-8" ?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
 <skin-addition>
  <skin-id>simple.desktop</skin-id>
  <style-sheet-name>acme/styles/acme-simple-desktop.css</style-sheet-name>
  <bundle-name>oracle.adfdemo.acme.faces.resource.AcmeSimpleDesktopBundle
  </bundle-name>
 </skin-addition>
</skins>
```

8. Add an image folder for the images used for the custom component. This folder should be under the `META-INF` directory. Place any images used by the custom component into this folder.

   For `tagPane`, the image folder is `/META-INF/acme/images`.

34.5 Deploying a Component Library

After creating the custom component library, you must create a deployable artifact that can be used by a web application.

34.5.1 How to Deploy a Component Library

Before you can build a Java archive (JAR) file, update the project's deployment profile by adding the many resources you created.

Before you begin:

You must have created the client-files for the component, as described in Section 34.3, "Developing for the Client-Side," and also the server-side files, as described in Section 34.4, "Developing for the Server-Side."

To create the JAR file for deployment:

1. In the Applications window, double-click the project.
2. In the Project Properties dialog, in the left pane, select **Compiler**.
3. On the right, ensure that all file types to be deployed are listed in the **Copy File Types to Output Directory** text field.

   Note: Some file types, such as `.css` and `.js` are not included by default. You will need to add these.

4. In the left pane, select **Deployment**.
5. On the right, under **Deployment Profiles**, select the ADF Library JAR file, and click **Edit**.
6. In the left pane, select **JAR Options**.
7. Verify the default directory path or enter a new path to store your ADF Library JAR file. Ensure that **Include Manifest File** is selected, and click **OK**.
8. To deploy, right-click the project and choose **Deploy** >***Project_name*** from the context menu. By default, the JAR file will be deployed to a deployment directory within the project directory.

34.6 Adding the Custom Component to an Application

After the component has been created and you have created an ADF Library, you can proceed to import it and use it in another application. However, before using it in an application under development, you should use it in a test application to ensure it works as expected. To do so, import the custom library into your test application. For procedures, see the "Adding ADF Library Components into Projects" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

After you add the library, you configure the web deployment descriptor to add a resource servlet mapping. When you use the component and run your test application, you may find you need to debug the component. Therefore, it helps to have logging and assertions enabled for the project.

Tip: Importing a library into an application allows the custom component to appear in JDeveloper's Components window.

34.6.1 How to Configure the Web Deployment Descriptor

You configured the component resource loader to assume a servlet resource mapping (for example, for the `tagPane` component, the mapping was `acme)`. Therefore, you must add the expected resource servlet mappings to the consuming application's `web.xml` file.

By default, MyFaces Trinidad skinning compresses the CSS classes when it normalizes CSS 3 into CSS 2. Turn off this compression while you are debugging the component. For a production deployment, toggle off this setting.

Before you begin:

It may be helpful to have an understanding of adding a custom component to an application. For more information, see Section 34.6, "Adding the Custom Component to an Application."

To configure the web.xml file:

1. In the Applications window, double-click the **web.xml** file.
2. In the overview editor, click the **Servlets** navigation tab.
3. Click the **Add** icon to add a new row to the table. Enter the following:
   - **Name**: Enter `resources`.
   - **Servlet Class**: Enter `org.apache.myfaces.trinidad.webapp.ResourceServlet`.
4. Below the table, click the **Servlet Mappings** tab.
5. Click the **Add** icon to add a new row to the table for a new servlet mapping. Enter a URI prefix. Resources beginning with this prefix will be handled by the servlet. For example, for the `tagPane` component, you might enter the prefix `/acme/*`.
6. To disable compression of the style sheet:
   a. Select **Application**.
   b. Click the **Add** icon for Context Initialization Parameters.
   c. For **Name**, enter `org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION` and for **Value** enter `true`.

34.6.2 How to Enable JavaScript Logging and Assertions

JavaScript debugging can be a difficult task. To help debug this dynamic language with no type checking, the ADF Faces JavaScript libraries provide a logging mechanism similar to Java logging. There is also an assertion strategy to make the client scripts more type safe. Both of these features are turned on using configuration parameters in the `web.xml` file. The logging and assertion routines are browser specific. The client JavaScript libraries will support Gecko, Internet Explorer, Opera, and Safari versions of browser agents. For more information, see Section A.2.3.4, "Resource Debug Mode."

Before you begin:

It may be helpful to have an understanding of adding a custom component to an application. For more information, see Section 34.6, "Adding the Custom Component to an Application."

You must have also configured the deployment descriptor. For more information, see Section 34.6.1, "How to Configure the Web Deployment Descriptor."

To turn on logging and assertion:

1. In the Applications window, double-click the **web.xml** file.
2. In the overview editor, click the **Application** navigation tab.
3. On the Application page, click the **Add** icon for the Context Initialization Parameters.
4. In the **Context Initialization Parameters** table, add the following parameter to turn on debugging:
   - **Name**: `org.apache.myfaces.trinidad.resource.DEBUG`
   - **Value**: `true`

   This setting prevents MyFaces Trinidad from setting the cache headers for resources like JavaScript. It prevents the browser from caching resources.
5. Add the following parameter to set the debug level for client side JavaScript.
   - **Name**: `oracle.adf.view.rich.LOGGER_LEVEL`
   - **Value**: `ALL`

   The valid values are `OFF`, `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST` and `ALL`. The default is `OFF`.
6. Add the following parameter to turn on client-side script assertions:
   - **Name**: `oracle.adf.view.rich.ASSERT_ENABLED`
   - **Value**: `true`

   This setting works together with logging. Toggling this switch to on will make debug information available to the browser. The assertions and logging are displayed differently, depending on the browser. For Internet Explorer, a child browser window will appear beside the active window. For FireFox with the Fire Bug plugin, the debug information will be available through the Fire Bug console.

34.6.3 How to Add a Custom Component to JSF Pages

Before you begin:

It may be helpful to have an understanding of adding a custom component to an application. For more information, see Section 34.6, "Adding the Custom Component to an Application."

You must have also configured the deployment descriptor. For more information, see Section 34.6.1, "How to Configure the Web Deployment Descriptor."

To add the custom component to a JSF page:

1. In the Applications navigator, double-click a JSP page.
2. In the source editor, add the TLD namespace to the root tag.

   For example, for the `tagPane` component, because the tag library's URI is: `http://adf-richclient-demo-acme`, you would add:

   ```
   xmlns:acme="http://oracle.adfdemo.acme"
   ```
3. Use the Components window to add the custom component to the page. Use the Properties window to set any attributes.

> **Tip:** If you are developing the application outside of JDeveloper, then on the page, use TLD short name and the component name. Also, add any values for attributes. For example, for the `tagPane`, you might add:

```
<acme:tagPane>
  <visible="true">
  <orderBy="alpha">
  <tagSelectionListener=#(tagBean.onTagSelect)
</tagPane>
```

34.6.4 What You May Need to Know About Using the tagPane Custom Component

If you wish to create the `tagPane` component as described in this chapter, and use it in an application, you will need to use backing beans to bind the custom component to the application components.

Example 34–24 shows the backing bean code that is used to bind the `tagPane` component to the File Explorer application.

Example 34–24 Backing Bean Logic for the tagPane Custom Component

```
public Map<String, Number> getTags()
 {
  if (_tags == null)
  {
    _tags = new TreeMap<String, Number>();
    List<FileItem> nameToFileItems = feBean.getDataFactory().getFileItemList();
   _doDeepTagCollection(_tags, nameToFileItems);
  }
  return _tags;
 }
 public void onTagSelect(TagSelectEvent event)
 {
  _selectedTag = event.getTag();
  CriteriaFileItemFilter criteria = new CriteriaFileItemFilter(_selectedTag);
  List<FileItem> nameToFileItems =    _feBean.getDataFactory().getFileItemList();
    if (_selectedTagFileItemList == null) {
       _selectedTagFileItemList = new ArrayList<FileItem>();
    else {
      _selectedTagFileItemList.clear();
     }
    _doDeepTagSearch(criteria, _selectedTagFileItemList, nameToFileItems);
    _selectedTagResultsTableModel = new SortableModel(_selectedTagFileItemList);

 }
```

35

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:

- Section 35.1, "About User Customization"
- Section 35.2, "Implementing Session Change Persistence"

35.1 About User Customization

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 35–1 shows the changes by component that provide default personalization capabilities:

Table 35–1 Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime

`panelBox`
`showDetail`
`showDetailHeader`
`showDetailItem`
	`disclosed`	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`	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`	Users can change the size of a panel, and that size will remain.
`panelSplitter`	`collapsed`	Users can collapse either side of the splitter. The collapsed state will remain as last configured by the user.
`panelSplitter`	`splitterPosition`	The position of the splitter in the panel will remain where last moved by user.
`richTextEditor`	`editMode`	The editor will display using the mode (either WYSIWYG or source) last selected by the user.
`calendar`	`activeDay`	The day considered active in the current display will remain the active day.
`calendar`	`view`	The view (day, week, month, or list) that currently displays activities will be retained.
`panelWindow` `dialog`	`contentHeight`	Users can change the height of a `panelWindow` or `dialog` popup component, and that height will remain.
`panelWindow` `dialog`	`contentWidth`	Users can change the width of a `panelWindow` or `dialog` popup component, and that width will remain.
`button` `link`	`windowHeight`	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.

Table 35–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
button link	windowWidth	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	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.
column	frozen	ADF Faces columns can be frozen so that they will not scroll. When a column's `frozen` attribute is set to `true`, all columns before that column (based on the `displayIndex` value) will not scroll. When you use the table with a `panelCollection` component, you can configure the table so that a button appears that allows the user to freeze a column. For more information, see Section 12.3.4, "How to Display a Table on a Page."
column	noWrap	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 `true` to `false`.
column	selected	The selected column is based on the column last selected by the user.
column	visible	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 `true` to `false`.
column	width	The width of the column will remain the same size as the user last set it.

Table 35–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
`table`	`filterVisible`	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:horizontal BarGraph`
`dvt:lineGraph`
`dvt:scatterGraph`
	`timeRangeMode`	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`	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.
`dvt:hierarchyViewer`	`layout`	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`	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.

Table 35–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
`dvt:map`	`srid`	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`	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.

`dvt:projectGantt`
`dvt:resource UtilizationGantt`
`dvt:schedulingGantt`
	`splitterPosition`	The position of the splitter in the panel will remain where last moved by user.

`dvt:timeAxis`	`scale`	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`	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.

35.1.1 User Customization Use Cases and Examples

You can configure an application so that the value of the attributes listed in Table 35–1 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*.

35.2 Implementing Session Change Persistence

In order for the application to persist user changes to the session, you must configure your project to enable customizations.

35.2.1 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**.

35.2.2 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. Example 35–1 shows the context parameter in the `web.xml` file.

Example 35–1 Context Parameter in web.xml Used for Change Persistence

```
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHANGE_PERSISTENCE</param-name>
  <param-value>session</param-value>
</context-param>
```

35.2.3 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.

35.2.4 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.

36 Adding Drag and Drop Functionality

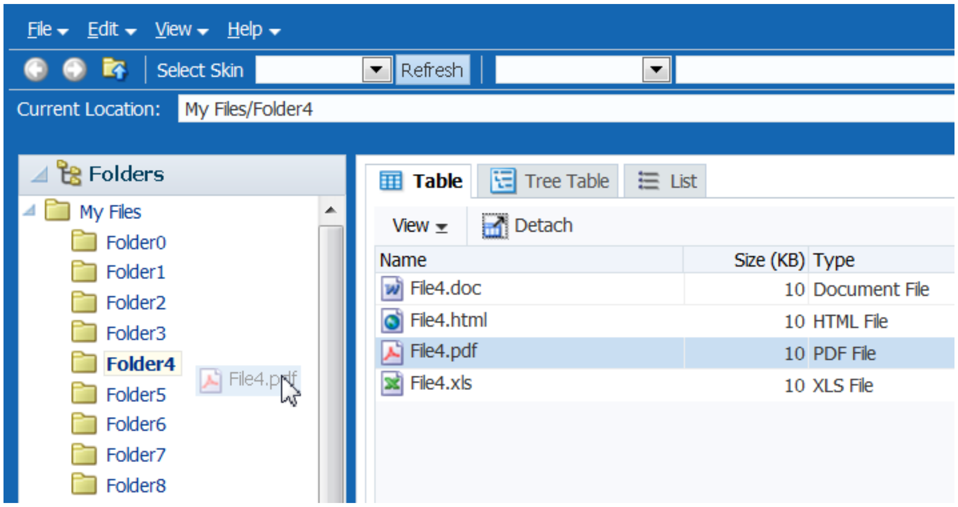
This chapter describes how to add drag and drop functionality to your pages, which allows users to drag the values of attributes or objects from one component to another, or allows users to drag and drop components. It describes how to add drag and drop functionality for attributes, objects, collections, 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:

36.1 About Drag and Drop Functionality

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 36–1.

Figure 36–1 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 36–1.
- 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.
- Dragging a marker in a DVT scatter or bubble graph to change its value.
- 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 36–1 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 36–1 Drag and Drop Scenarios

Scenario	Source	Target
Dragging an attribute value	An attribute value on a component	An attribute value on another component, as long as it is the same object type
	Tag: `attributeDragSource`	**Tag**: `attributeDropTarget`
Dragging an object from one component to another	Any component	Any component
	Tag: `attributeDragSource`	**Tag**: `dropTarget`
Dragging an item from one collection and dropping it into another	`table`, `tree`, and `treeTable` components	`table`, `tree`, and `treeTable` components
	Tag: `dragSource`	**Tag**: `collectionDropTarget`
Dragging a component from one container to another	Any component	Any component
	Tag: `componentDragSource`	**Tag**: `dropTarget`
Dragging a calendar activity from one start time or date to another	`calendarActivity` object	`calendar` component
	Tag: None needed	**Tag**: `calendarDropTarget`
Dragging a `panelBox` component into a `panelDashboard` component.	`panelBox` component	`panelDashboard` component
	Tag: componentDragSource	**Tag**: `dataFlavor`
Dragging a panelBox component out of a `panelDashboard` component.	panelBox component in a `panelDashboard` component	Any component
	Tag: componentDragSource	**Tag**: `dropTarget`
Dragging a marker in a DVT graph	`graph` component	`graph` component
	Tag: `dragSource`	**Tag**: `dropTarget`

Table 36–1 (Cont.) Drag and Drop Scenarios

Scenario	Source	Target
Dragging an object from a DVT Gantt chart and dropping it on another component	Gantt chart	Any component
	Tag: `dragSource`	**Tag**: `dropTarget`
Dragging a node from a DVT hierarchy viewer, sunburst, or treemap and dropping it on another component	`hierarchyViewer`, `sunburst`, or `treemap` component	Any component
	Tag: `dragSource`	**Tag**: `dropTarget`
Dragging an event from a timeline and dropping it into a collection component	`timeline` components	`table`, `tree`, and `treeTable` components
	Tag: `dragSource`	**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.

36.1.1 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 Section 3.6, "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 Chapter 6, "Handling Events."

36.2 Adding Drag and Drop Functionality for Attributes

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.

36.2.1 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 Section 36.2, "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 Section 36.1.1, "Additional Functionality for Drag and Drop."

To add drag and drop functionality for attributes:

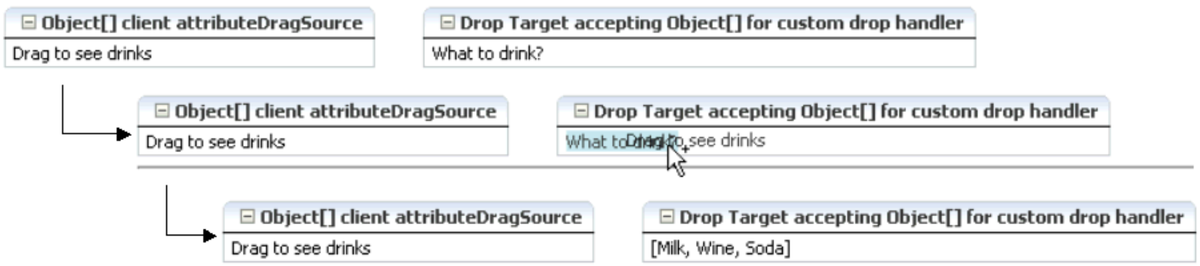
1. In the Components window, from the Operations panel, in the Drag and Drop group, drag and drop an **Attribute Drop Target** as a child to the target component on the page.
2. In the Insert Attribute Drop Target dialog, use the **Attribute** dropdown to select the attribute that will be populated by 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.

36.3 Adding Drag and Drop Functionality for Objects

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 36–2.

Figure 36–2 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 Section 36.8, "Adding Drag and Drop Functionality for DVT Components.".

36.3.1 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 Section 36.3, "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 Section 36.1.1, "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 Section 3.6, "Creating and Using Managed Beans."

   > **Tip:** You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see Section 36.3.3, "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`.

   Example 36–1 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`.

Example 36–1 Code for a dropTarget tag

```
<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.

Example 36–2 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.

Example 36–2 Event Handler Code for a dropListener

```
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. Example 36–3 shows the code for an `outputText` component that is the source of the drag and drop operation.

Example 36–3 Code for a Drag Source

```
<af:outputText value="Drag to see drinks">
  <af:clientAttribute name="drinks" value="#{myBean.drinks}"/>
  <af:attributeDragSource attribute="drinks"/>
</af:outputText>
```

36.3.2 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 Section 6.4.3, "What You May Need to Know About Marshalling and Unmarshalling Data.".

36.3.3 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. Example 36–4 shows a client handler that uses the logger to print a message.

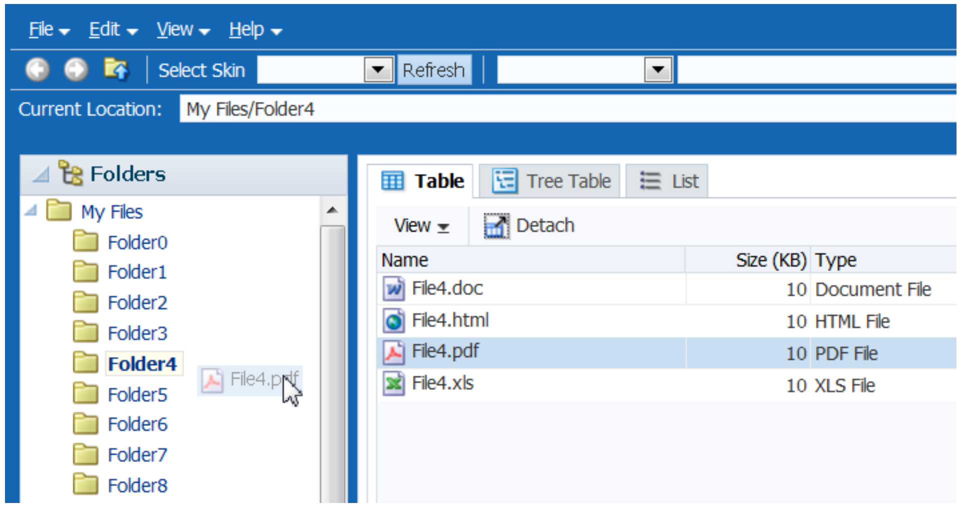
Example 36–4 clientDropListener Handler

```
<script>
/**
 * Shows a message.
 */
function showMessage()
{
  AdfLogger.LOGGER.logMessage(AdfLogger.ALL, "clientDropListener handler,
    copying...");
  return AdfDnDContext.ACTION_COPY;
}
</script>
```

36.4 Adding Drag and Drop Functionality for Collections

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 36–3 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 36–3 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. For more information, see Section 36.4.2, "What You May Need to Know About the dragDropEndListener".

36.4.1 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 Section 36.4, "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 Section 36.1.1, "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 **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.

   Example 36–5 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.

Example 36–5   Event Handler Code for a dropListener for a Collection

```
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}"/>
```

36.4.2 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. Example 36–6 shows a handler for a `dragDropEndListener` attribute

Example 36–6 Handler for dragDropEndListener

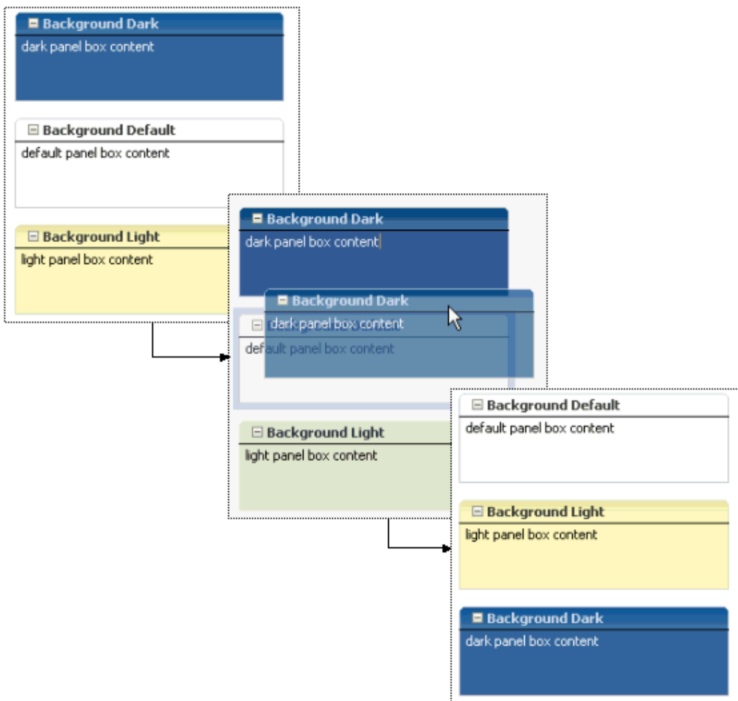
```
public void endListener(DropEvent dropEvent)
{
  Transferable transferable = dropEvent.getTransferable();

  // 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()
);
```

36.5 Adding Drag and Drop Functionality for Components

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 36–4 shows the darker panelBox component being moved from being the first child component of the panelGrid component to the last.

Figure 36–4 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. For more information, see Section 36.6, "Adding Drag and Drop Functionality Into and Out of a panelDashboard Component."

36.5.1 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 Section 36.5, "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 Section 36.1.1, "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.

   Example 36–7 shows the `handleComponentMove` event handler on the `DemoDropHandler.java` managed bean used by the `componentDragSource` JSF page in the demo application.

Example 36–7 Event Handler Code for a dropListener That Handles a Component Move

```
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"/>
   ```

36.6 Adding Drag and Drop Functionality Into and Out of a panelDashboard Component

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.

36.6.1 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 Section 36.6, "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 Section 36.1.1, "Additional Functionality for Drag and Drop."

Before you begin:

1. Create a `panelDashboard` component. For more information, see Section 9.8, "Arranging Contents in a Dashboard."
2. Create another component outside of the `panelDashboard` that contains `panelBox` components. For more information about `panelBox` components, see Section 9.9.3, "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.

36.6.2 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 Section 36.6, "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 Section 36.1.1, "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.

   Example 36–8 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.

Example 36–8 Event Handler Code for a dropListener to Move a panelBox Out of a panelDashboard

```
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.

36.7 Adding Drag and Drop Functionality to a Calendar

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.

36.7.1 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 Section 36.7, "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 Section 36.1.1, "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 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[]`.

36.7.2 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.

36.8 Adding Drag and Drop Functionality for DVT Components

You can configure drag and drop functionality for the following DVT components:

- Bubble and scatter graphs
- 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.

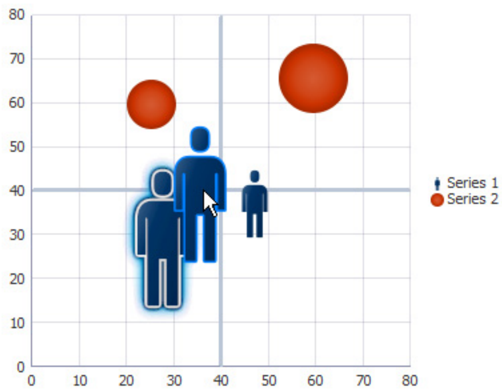
As with dragging and dropping objects or collections, you must also implement a `dropListener` handler to respond to the drop requests. The object of the drop event is called the `Transferable`, which contains the payload of the drop. Your listener must access the `Transferable` object, and from there, 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.

36.8.1 Adding Drag and Drop Functionality for DVT Graphs

You can configure drag and drop for the DVT bubble and scatter graphs, which allows the user to change the value of a marker by repositioning it. When you want users to be able to drag and drop in a graph, you use the `dragSource` and `dropTarget` tags. Additionally, you use the `DataFlavor` object to determine the valid Java type of the sources for the drop target, in this case a `GraphSelectionSet` object. You also must implement any required functionality in response to the drag and drop action.

For example, you might have a `bubbleGraph` component and you want the user to be able to drag a human marker to adjust the performance rating of an employee, as shown in Figure 36–5.

Figure 36–5 Dragging and Dropping an Object



The bubbleGraph component contains both a dragSource tag and a dropTarget tag. You also use a dataFlavor tag to determine the type of object being dropped.

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.

36.8.1.1 How to Add Drag and Drop Functionality for a DVT Graph

To add drag and drop functionality, first add source and target tags to the graph. Then implement the event handler method that will handle the logic for the drag and drop action. For information about what happens at runtime, see Section 36.3.2, "What Happens at Runtime: How to Use Keyboard Modifiers."

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 graph 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).
3. In the Insert Data Flavor dialog, enter oracle.adf.view.faces.bi.component.graph.GraphSelectionSet, which is the class for the object that can be dropped onto the target. This entry will be used to create a dataFlavor tag, which determines the type of object that can be dropped onto the target.
4. 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 graphs that both accept a GraphSelectionSet object, GraphA and GraphB. You also have two sources, both of which are GraphSelectionSet objects. By setting a discriminant value on GraphA with a value of alpha, only the GraphSelectionSet source that provides the discriminant value of alpha will be accepted.
5. In the Structure window, select the dropTarget tag. 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 (copy and paste), MOVE (cut and

paste), or `LINK` (copy and paste as a link). If you do not specify a value, the drop target will use `COPY`.

For example, to change the human marker to another value in Figure 36–5, the drop target's **Actions** field is set to `MOVE`.

6. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the graph component.

7. In the Structure window, select the `dragSource` tag. In the Properties window, in the **Actions** field, enter a list of the operations that the drag source will support, separated by spaces. Allowable values are: `COPY`, `MOVE`, or `LINK`. If you do not specify a value, the drag source will use `COPY`. Add any needed discriminant, as configured for the `dataFlavor` tag.

   For example, to change the human marker to another value in Figure 36–5, the drag source's **Actions** field is set to `MOVE`.

8. 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.

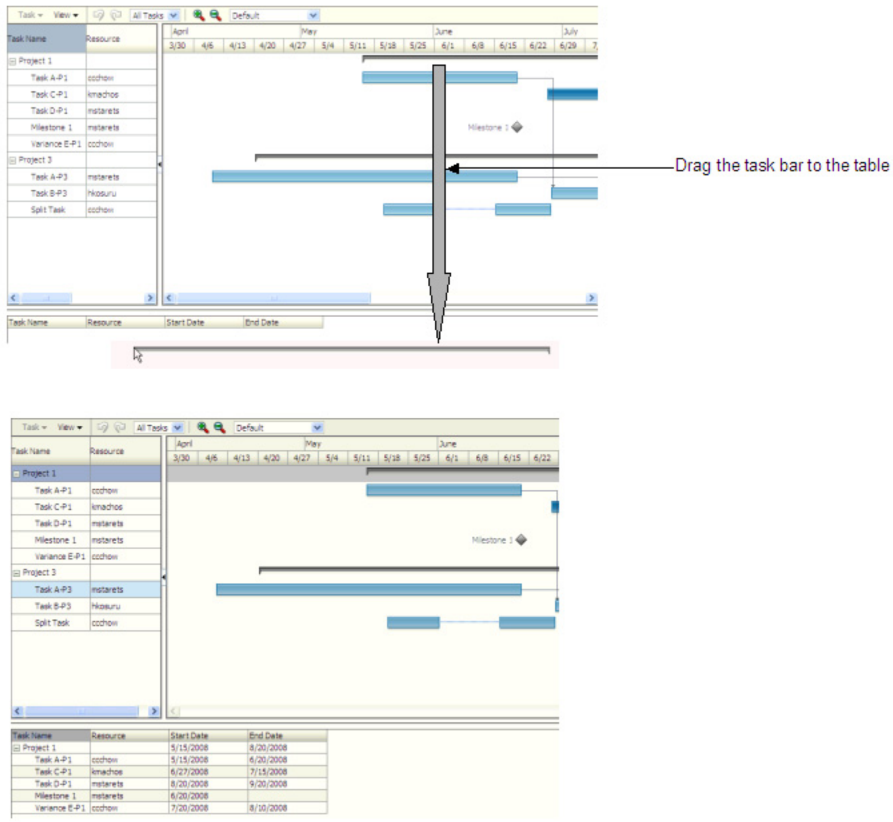
   This method must take a `DropEvent` event as a parameter and return a `DnDAction` object, which is the action that will be performed when the source is dropped, in this case `DnDAction.MOVE`. This method should check the `DropEvent` event to determine whether or not it will accept the drop. If the method 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 of the `dataFlavor` object, in this case `oracle.adf.view.faces.bi.component.graph.GraphSelectionSet`.

36.8.2 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 36–6.

Figure 36–6 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.

36.8.2.1 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 Section 36.3.2, "What Happens at Runtime: How to Use Keyboard Modifiers." For information about using the clientDropListener attribute, see Section 36.3.3, "What You May Need to Know

About Using the ClientDropListener."

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 Section 36.3.3, "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`.

   Example 36–9 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.

Example 36–9 JSP Code for a dropTarget tag

```
<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.

   Example 36–10 shows a handler method that copies a `TaskDragInfo` object from the event payload and assigns it to the component that initiated the event.

Example 36–10 Event Handler Code for a dropListener

```
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.

36.8.3 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.

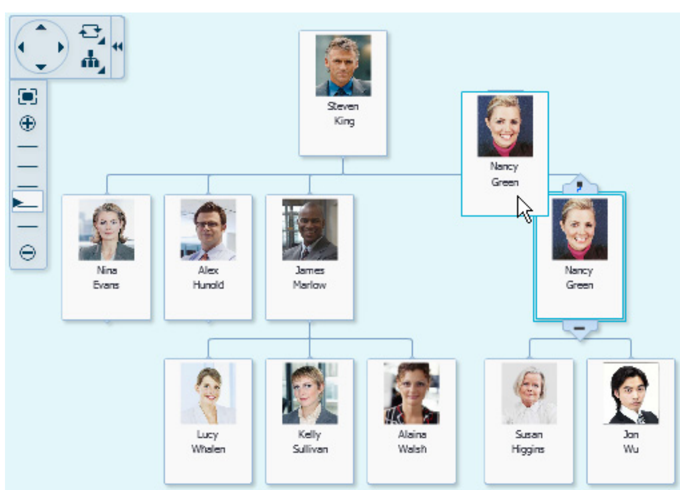
36.8.3.1 Drag and Drop Example for DVT Hierarchy Viewers

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

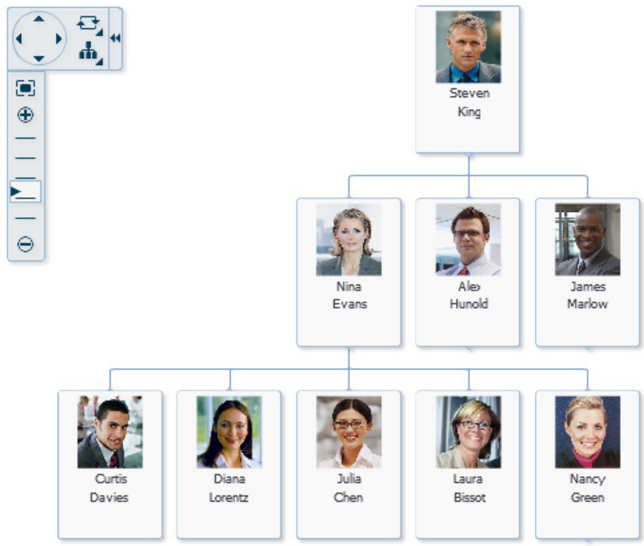
Figure 36–7 shows a hierarchy viewer configured to allow drags and drops within itself. If you click and hold a node for more than one-half second, you can drag it to the background to make it another root in the hierarchy or drag it to another node to add it as a child of that node.

Figure 36–7 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 36–8 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 36–8 Hierarchy Viewer After Node Drag to Another Node

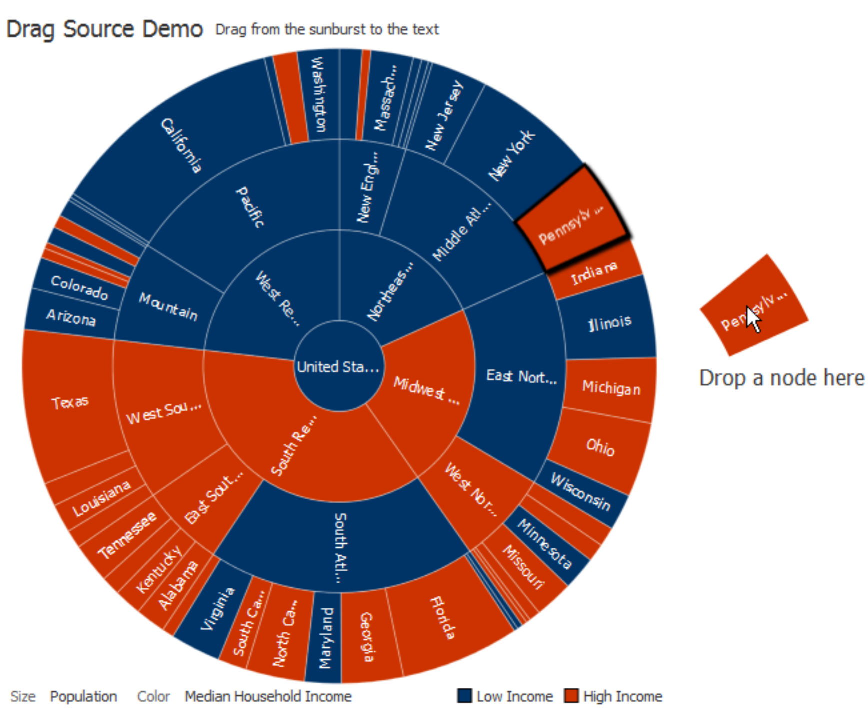


36.8.3.2 Drag and Drop Example for DVT Sunbursts

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 36–9 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 36–9 Sunburst Configured as a Drag Source

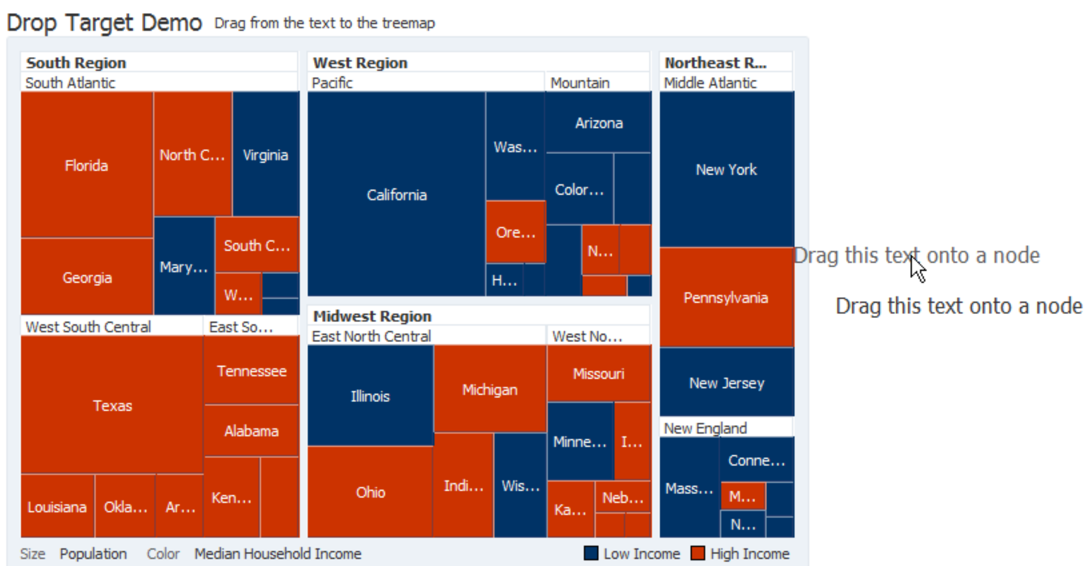


36.8.3.3 Drag and Drop Example for DVT Treemaps

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 36–10 shows a treemap configured as a drop target. In this example, the drag source is an `af:outputFormatted` component.

Figure 36–10 Treemap Configured as a Drop Target



36.8.3.4 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 Section 36.3.2, "What Happens at Runtime: How to Use Keyboard Modifiers." For information about using the `clientDropListener` attribute, see Section 36.3.3, "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 Section 36.1, "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 Chapter 22, "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 Section 1.3, "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:
   1. 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.
   2. 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 6).

      Tip: You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see Section 36.3.3, "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 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.
   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`.

      Example 36–11 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.

Example 36–11 JSP Code for a dropTarget tag on a DVT Treemap Component

```
<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>
```

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.

   Example 36–12 shows a handler method that copies a `java.lang.Object` from the event payload and assigns it to the component that initiated the event.

Example 36–12 Sample Drop Listener for a DVT Treemap

```
// 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:

   1. In the Components window, from the Operations panel, drag and drop a **Drag Source** as a child to the DVT component.

   2. With the `dragSource` tag selected, in the Properties window, set the allowed Actions and any needed discriminant, as configured for the target.

   Example 36–13 shows the JSP code for a treemap configured as a drag source. Note that all actions (`COPY`, `MOVE`, and `LINK`) are permitted.

Example 36–13 JSP Sample Code for a dragSource Tag on a DVT Treemap

```
<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:

   1. 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.

   2. 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.

   3. 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`.

   4. In the Structure window, right-click the `af:dropTarget` component and choose **Go to Properties**.

   5. 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.

Example 36–14 shows the sample code for an af:outputFormatted component configured to allow dragging from a treemap.

Example 36–14 JSP Sample Code for Dragging Data from a Treemap to an af:outputFormatted Component

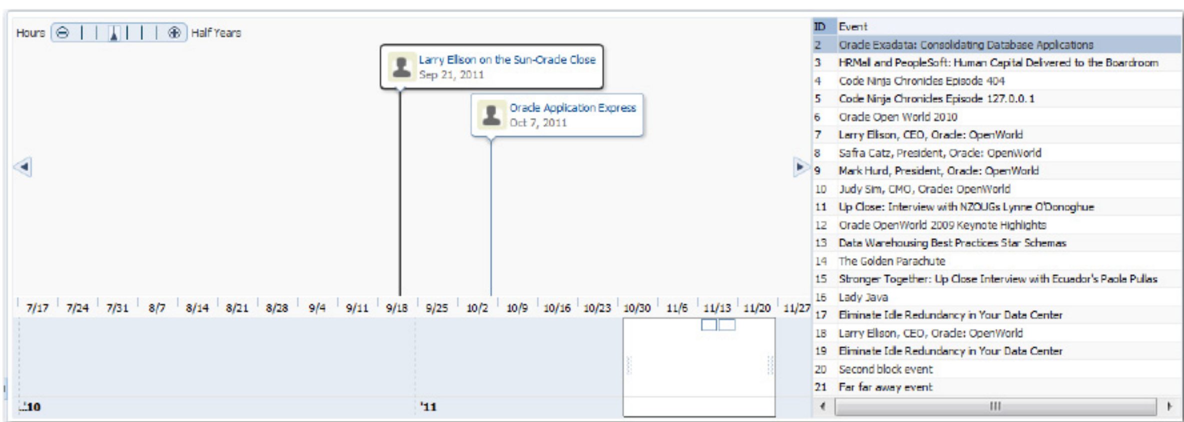
```
<af:outputFormatted value="#{treemap.dropText}" id="of1">
  <af:dropTarget dropListener="#{treemap.fromDropListener}">
    <af:dataFlavor flavorClass="org.apache.myfaces.trinidad.model.RowKeySet"/>
  </af:dropTarget>
</af:outputFormatted>
```

36.8.4 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 36–11 shows a timeline configured to allow drags and drops between the events in a timeline and a row in a table.

Figure 36–11 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 Section 36.3.2, "What Happens at Runtime: How to Use Keyboard Modifiers." For information about using the clientDropListener attribute, see Section 36.3.3, "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 Section 36.1, "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 Chapter 22, "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 Section 1.3, "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.

   Example 36–15 shows the sample drop listener and supporting methods for the timeline displayed in Figure 36–11.

Example 36–15 Sample Drop Listener for a Timeline

```
// 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 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 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.

      Example 36–16 shows the sample drop listener for the timeline displayed in Figure 36–11. This example uses the same imports and helper methods used in Example 36–15, and they are not included here.

Example 36–16 Sample Drop Listener for a Table Using a Timeline as a Drag Source

```
//Drop Listener
public DnDAction handleDropOnTable(DropEvent event)
{
  Integer _dropSite = (Integer)event.getDropSite();
  Transferable _transferable = event.getTransferable();
  RowKeySet _rowKeySet = _transferable.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.

Example 36–17 shows the JSF page sample code for the ADF Faces demo application illustrated in Figure 36–11. For additional information about the `af:table` component, see Chapter 12, "Using Tables, Trees, and Other Collection-Based Components."

Example 36–17 Sample Code for Timeline Drop Target and Drag Source

```
<dvt:timeline id="tl1" 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>
  <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:timelineSeries>
  <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>
```

37

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:

37.1 About Using Different Output Modes

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.

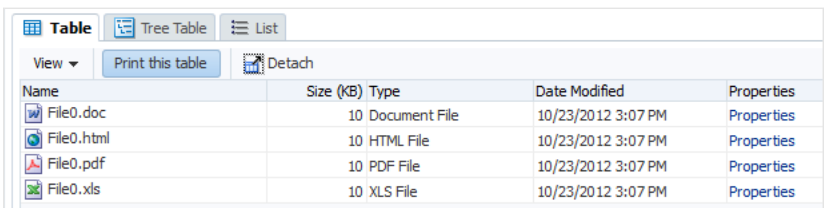
For email support, ADF Faces provides an API that can be used to convert a page to one that is suitable for display in the Microsoft Outlook 2007, Mozilla Thunderbird 10.0.5, or Gmail email clients.

> **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()`.

37.1.1 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 37–1.

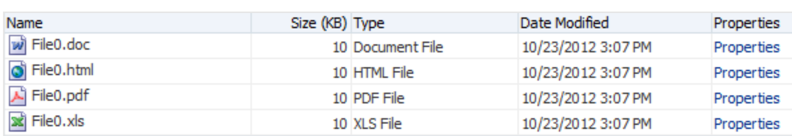
Figure 37–1 Button to Print Part of a Page



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

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 37–2.

Figure 37–2 Printable Version of the Page



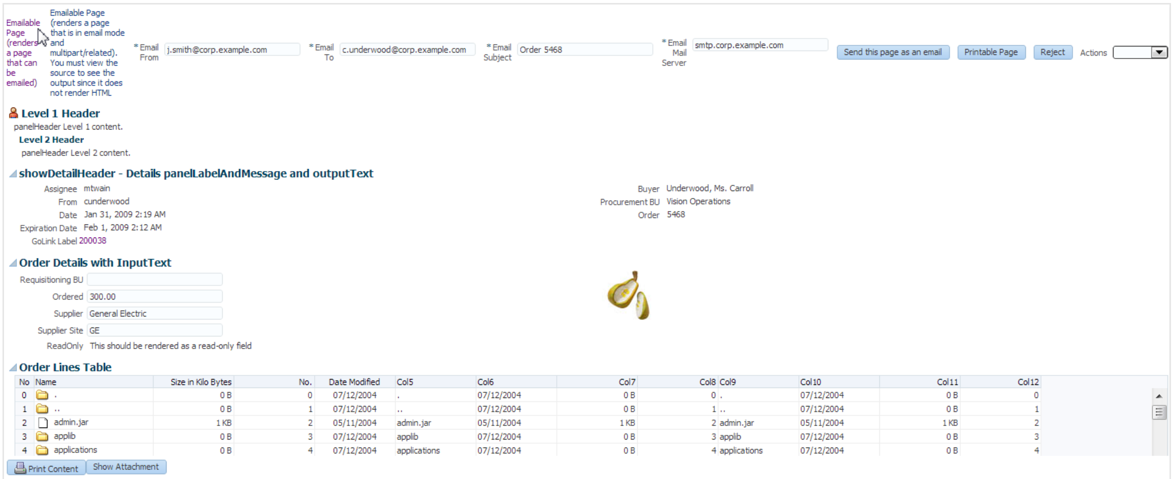
Name	Size (KB)	Type	Date Modified	Properties
File0.doc	10	Document File	10/23/2012 3:07 PM	Properties
File0.html	10	HTML File	10/23/2012 3:07 PM	Properties
File0.pdf	10	PDF File	10/23/2012 3:07 PM	Properties
File0.xls	10	XLS File	10/23/2012 3:07 PM	Properties

Only the contents of the table are displayed for printing. All extraneous components, such as tabs, toolbar, 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.

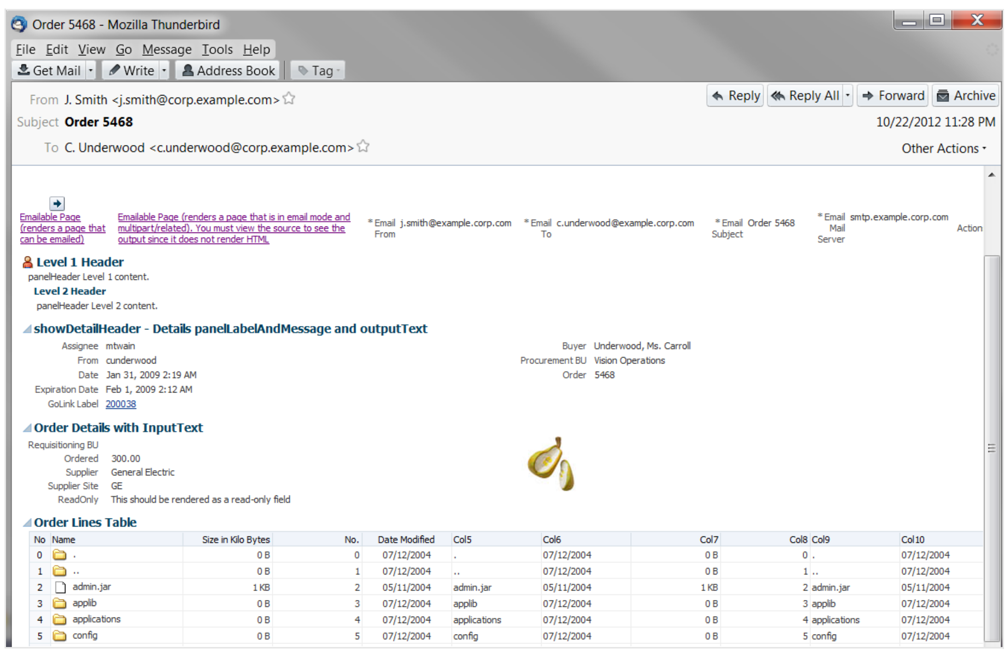
Say you have a page that displays a purchase order, as shown in Figure 37–3.

Figure 37–3 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 37–4 shows the page as it appears in an email client.

Figure 37–4 Page in an Email Client



37.2 Displaying a Page for Print

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 in the print version (such as buttons, tabs, 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.

37.2.1 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 Section 37.2, "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 Section 20.3.1, "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.

37.3 Creating Emailable Pages

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.

37.3.1 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 Section 37.3, "Creating Emailable Pages."

To create an emailable page:

1. Insert a command component onto the page to be emailed. For more information, see Chapter 20, "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 and emails the page.

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.r
ender.emailContentType=multipart/related
```

The framework will convert the HTML to MIME (multipart/related) and inline the images so the email can be viewed offline.

3. Select the command component, and in the Properties window, set the method or service as the value for **ActionListener**.

37.3.2 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.

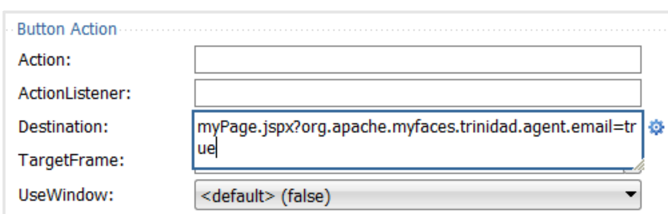
To test an emailable page:

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 37–5 Setting the Destination Attribute



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 your 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. Example 37–1 shows how you might specify the border on a table rendered in email.

Example 37–1 Skin for Emailable Page

```
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 Chapter 31, "Customizing the Appearance Using Styles and Skins."

37.3.3 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.
- Remove the CSS link from the HTML.
- Convert all relative links to absolute links.
- Render images with absolute URLs.

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.emailCont
entType=multipart/related
```

38

Using the Active Data Service with an Asynchronous Backend

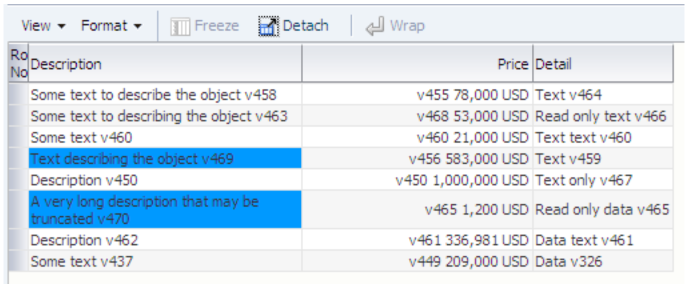
This chapter describes how to register an asynchronous backend with Active Data Service (ADS) to provide real-time data updates to ADF Faces components.

This chapter includes the following sections:

38.1 About the Active Data Service

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 38–1.

Figure 38–1 Table Displays Updated Data as Highlighted



38.1.1 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 to work with active data:

- `activeImage`
- `activeOutputText`
- `table`

  Note that filtering on a table that will be using active data is not supported. Once a table is filtered at runtime, active data cannot be displayed. Currently, ADS supports `table` components with the `outputText` component contained within a column; other components are not supported inside the table column.
- `tree`
- `treeTable`
- `graph`, `gauge`, `mapPointTheme`, `pivotTable`, `pivotFilterBar`, `sunburst`, and `treemap` ADF Data Visualization components, as identified in Section 22.2.3, "Active Data Support"

  Note that ADS does not support active data for all `graph` component types. For a list of supported graph types, see Section 23.6.1, "How to Configure Databound Graph Components to Display Active Data."

For details about the active data service framework and important configuration information, see *Developing Fusion Web Applications with Oracle Application Development Framework*.

38.2 Process Overview for Using Active Data Service

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 *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 *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 the ADF component that you use to capture and display the pushed data.

38.3 Implementing the ActiveModel Interface in a Managed Bean

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
- `ActiveDataModelDecorator` class (for use with graphs)
- `ActiveGeoMapDataModelDecorator` class
- `ActiveGaugeDataModelDecorator` 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 Example 38–1):

1. Create a Java class that extends the decorator class appropriate for your component.

   Example 38–1 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.

   Example 38–1 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.

   Example 38–1 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.

Example 38–1 Extend the Decorator Class

```
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. Example 38–2 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.

Example 38–2 Register as a Managed Bean

```
...
<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>
```

38.4 What You May Need to Know About Maintaining Read Consistency

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. Example 38–3 shows how you can use your implementation of the `ActiveDataModel` class to maintain read consistency.

38.5 Passing the Event Into the Active Data Service

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.

Example 38–3 shows the `notifyDataChange()` method of the model passes the `Event` object to the ADS framework, by placing the object into the `fireActiveDataUpdate()` method.

Example 38–3 Pass the Event Object into ADS

```
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 listerners
     * 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();
}
```

38.6 Registering the Data Update Event Listener

You need to register a data change listener for data change events from the backend. Example 38–4 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.

Example 38–4 Register the Data Update Event Listener

```
...
<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>
```

38.7 Configuring the ADF Component to Display Active 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.

Example 38–5 shows the expression language used on the `table` component `value` attribute to receive the pushed data.

Example 38–5 Display the Active Data

```
...
<f:view>
   <af:document id="d1">
      <af:form id="f1">
        <af:panelStretchLayout topHeight="50px" id="psl1">
          <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>
```

Part VII

Appendixes

Part VII contains the following appendixes:

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:

A.1 About Configuring ADF Faces

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 more information about the relationship between Trinidad and ADF Faces, see Chapter 1, "Introduction to ADF Faces."

A.2 Configuration in web.xml

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, custom tag library location, 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 more information about the required elements, see Section A.2.2, "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 Section A.2.3, "What You May Need to Know About ADF Faces Context Parameters in web.xml."

For information about configuring `web.xml` outside of ADF Faces, see *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

A.2.1 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.

Example A–1 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 Section A.2.2, "What You May Need to Know About Required Elements in web.xml."

Example A–1 Generated web.xml File

```
<?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 Section A.2.3, "What You May Need to Know About ADF Faces Context Parameters in web.xml."

A.2.2 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 more information about state-saving context parameters, see Section A.2.3, "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.

A.2.3 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>
```

The following context parameters are supported for ADF Faces.

A.2.3.1 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.
- `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 `off`.

A.2.3.2 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.

A.2.3.3 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.

A.2.3.4 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) can 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`.

A.2.3.5 User Customization

For more information about enabling and using session change persistence, see Chapter 35, "Allowing User Customization on JSF Pages."

A.2.3.6 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 more information about RUEI, see the "Enabling the Application for Real User Experience Insight and End User Monitoring" section in *Developing Fusion Web Applications with Oracle Application Development Framework*.

A.2.3.7 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.

A.2.3.8 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).

A.2.3.9 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.

> **Performance Tip:** Compression will affect performance. In a production environment, set this parameter to false.

A.2.3.10 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.

A.2.3.11 Test Automation

When you set the oracle.adf.view.rich.automation.ENABLED parameter to true and when the component ID attribute is null, the component testId attribute is used during automated testing to ensure that the ID is not null. The testId is an attribute only on the tag. It is not part of the Java component API. Note this context 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.

Caution: When the test automation context parameter is set to `true`, 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. For more information, see Section A.2.3.18, "Framebusting."

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 that 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=true`. Running your application in an application server instance with test automation enabled overrides the `web.xml` file context parameter setting of the deployed application.

A.2.3.12 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.

A.2.3.13 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.

A.2.3.14 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.

A.2.3.15 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. For more information, see Section 8.5, "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 more information about `partialSubmit`, see Section 6.1.1, "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 more information about `partialSubmit`, see Section 6.1.1, "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.

A.2.3.16 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.

A.2.3.17 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 more information about using and configuring JavaScript partitioning, see Section 4.9, "JavaScript Library Partitioning."

A.2.3.18 Framebusting

Use the `org.apache.myfaces.trinidad.security.FRAME_BUSTING` context parameter to use framebusting in your application. *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.

For example, say 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;
```

If you configure your application to use framebusting by setting the 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 `DoaminBpage1` 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.

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.

A.2.3.19 Version Number Information

Use the `oracle.adf.view.rich.versionString.HIDDEN` parameter to determine whether or not to display version information an 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 Example A–2.

Example A–2 Version Information in the HTML

```
</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.

A.2.3.20 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.

A.2.3.21 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 more information about defining caching rules, see Section A.4.2, "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 more information about settings in adf-settings.xml, see Section A.5.2, "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

Example A–3 shows a sample ACF servlet definition.

Example A–3 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.

A.2.3.22 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

A.2.3.23 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. For more information, see Section A.2.3.2, "Debugging."

A.2.3.24 Session Timeout Warning

Use the `oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_TIMEOUT` context parameter to set the number of seconds prior to the session timeout when a warning dialog is displayed. When the warning is displayed, the user is given the opportunity to extend the session. If the session is not extended, a final alert dialog is displayed notifying the user of the timeout. When this dialog is shown, the content in the page underneath the dialog is no longer visible. Depending on the application security configuration, the user may be redirected to the log in page when the session expires.

The default value of this parameter is 120 seconds. If the value of `WARNING_BEFORE_TIMEOUT` is set to less than 120 seconds, if client state saving is used for the page, or if the session has been invalidated, the feature is disabled.

Example A–4 shows configuration of the warning dialog to display at 120 seconds before the timeout of the session.

Example A–4 Configuration of Session Timeout Warning

```
<context-param>
    <param-name>oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_
        TIMEOUT</param-name>
    <param-value>120</param-value>
</context-param>
```

A.2.3.25 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.

A.2.3.26 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.

A.2.3.27 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.

A.2.3.28 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`.

A.2.3.29 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.

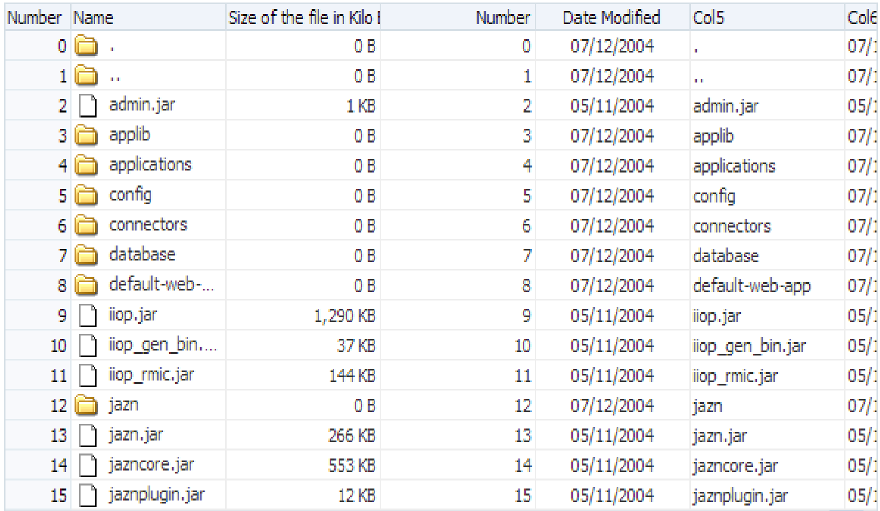
Set oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS parameter to auto when you want both layout components and tables to always stretch when the parent component allows stretching.

A.2.3.30 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.

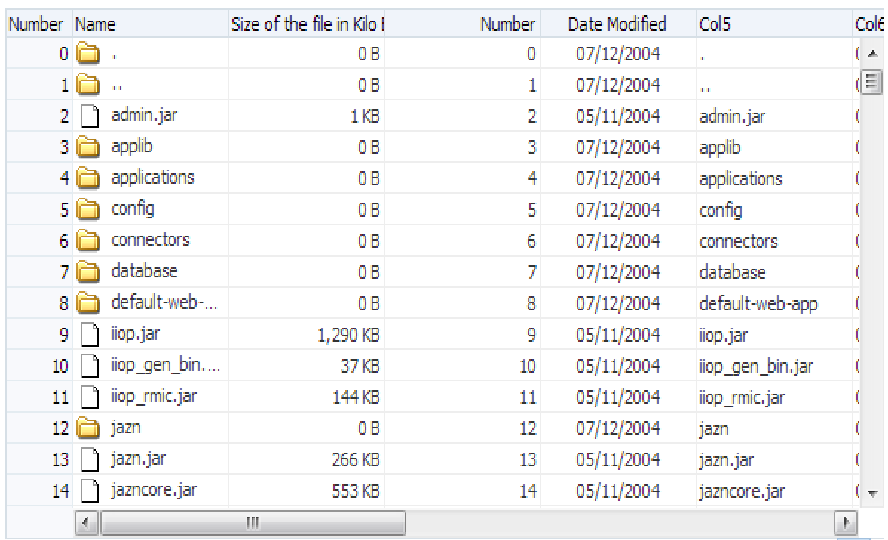
Figure A–2 Scrollbars are Hidden



Number	Name	Size of the file in Kilo I	Number	Date Modified	Col5	Col6
0	.	0 B	0	07/12/2004	.	07/1
1	..	0 B	1	07/12/2004	..	07/1
2	admin.jar	1 KB	2	05/11/2004	admin.jar	05/1
3	applib	0 B	3	07/12/2004	applib	07/1
4	applications	0 B	4	07/12/2004	applications	07/1
5	config	0 B	5	07/12/2004	config	07/1
6	connectors	0 B	6	07/12/2004	connectors	07/1
7	database	0 B	7	07/12/2004	database	07/1
8	default-web-...	0 B	8	07/12/2004	default-web-app	07/1
9	iiop.jar	1,290 KB	9	05/11/2004	iiop.jar	05/1
10	iiop_gen_bin....	37 KB	10	05/11/2004	iiop_gen_bin.jar	05/1
11	iiop_rmic.jar	144 KB	11	05/11/2004	iiop_rmic.jar	05/1
12	jazn	0 B	12	07/12/2004	jazn	07/1
13	jazn.jar	266 KB	13	05/11/2004	jazn.jar	05/1
14	jazncore.jar	553 KB	14	05/11/2004	jazncore.jar	05/1
15	jaznplugin.jar	12 KB	15	05/11/2004	jaznplugin.jar	05/1

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 I	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.

A.2.3.31 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

---

A.2.4 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.

- `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION`: Specifies whether JSP and CSS files require a restart in order to see changes at runtime. By default, set to `false`. Set to `true` if you want to be able to view changes without restarting the server.

A.3 Configuration in faces-config.xml

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 more information about the `faces-config.xml` file, see the Java EE 6 tutorial (`http://download.oracle.com/javaee/index.html`).

A.3.1 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). For more information, see Section 3.2, "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 Example A–5.

Example A–5 ADF Default Render Kit Configuration in faces-config.xml

```
<?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. For more information, see the "Getting Started with ADF Task Flows" chapter of *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.

A.4 Configuration in adf-config.xml

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.

A.4.1 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**.
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 Example A–6.

Example A–6 XML for adf-config.xml File

```
<?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.

A.4.2 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 Section A.2.3.21, "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. Example A–7 shows the syntax for defining caching rules in `adf-config.xml`.

Example A–7 ACF Caching Rule Syntax

```
<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
`<cache>`	Specifies whether or not the object must be cached in the web cache. A value of `false` will ensure the object is never cached. The default is `true`.
`<duration>`	Defines the duration in seconds for which the object will be cached in the web cache. The default is `300` seconds.
`<agent-caching>`	Specify a value of `true` to use a browser cache in the absence of a web cache.
`<agent-duration>`	Defines the duration in seconds for which the object is cached in a browser cache. The default is `-1`. If `<agent-caching>` is `true` and `<agent-duration>` is not defined, then the value for `<duration>` is used instead.
`<compress>`	Specifies whether or not the object cached in the web cache must be compressed. The default value is `true`.
`<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 `<caching-rule>` 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>`
	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.
	Each `<vary-element>` is defined by:
	■ `<vary-name>`: Valid values are `cookieName` for the name of the cookie whose value the response varies on, or `headerName` 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 `cookie` or `header`.
	The web cache automatically versions request parameters. Multiple version of an object will be stored in web cache based on the request parameter.

A.4.3 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.
- `disable`: 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.

Example A–8 shows the syntax for application-wide disabling of Flash in `adf-config.xml`.

Example A–8 Flash Disabled 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.

A.4.4 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 Example A–9:

Example A–9 Default Image Reference

```
<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 Example A–10:

Example A–10 Image Reference from a CDN Using an Absolute URI

```
<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 Example A–11:

Example A–11 EL Binding to a CDN Base URI

```
<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, Example A–12 specifies that all `png` images in the images directory should be rewritten to include the `http://mycdn.com` prefix.

Example A–12 Specifying a CDN Prefix in adf-config.xml

```
<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 Example A–13 applies the `http://mycdn.com/` prefix to all images defined by ADF Faces components:

Example A–13 Adding a Prefix to Images

```
<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 (for more information, see Section A.4.1, "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 `ExteralContext.encodeResourceURL()`. If a URI matches, the prefix specified in the enclosing `<base-resource-uri>` element is applied.

Example A–14 shows how an application might be configured to use a CDN.

Example A–14 CDN URI Elements

```
<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>
```

A.4.4.1 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 a 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.

A.4.4.2 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 `-Doracle.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 `-Doracle.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&accessibi
   lity=default
   ```

A.5 Configuration in adf-settings.xml

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.

A.5.1 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 more information about configurations in this file, see Section A.5.2, "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**.
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 Example A–15, using the correct settings for your web application root.

Example A–15 XML for adf-settings.xml File

```
<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. For more information, see Section A.5.2, "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.

A.5.2 What You May Need to Know About Elements in adf-settings.xml

The following configuration elements are supported in the `adf-settings.xml` file.

A.5.2.1 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.

Example A–16 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.

Example A–16 Help Provider Registration

```
<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>
```

A.5.2.2 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`. Example A–17 shows the `adf-settings.xml` file for the ADF Faces JAR.

Example A–17 ADF Faces adf-settings.xml File

```
<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>
```

A.6 Configuration in trinidad-config.xml

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`. For more information, see Section A.6.3, "What You May Need to Know About Configuring a System Property."

A.6.1 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. Example A–18 shows a starter trinidad-config.xml file.

Example A–18 Starter trinidad-config.xml File Created by JDeveloper

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://xmlns.oracle.com/trinidad/config">

  <skin-family>skyros</skin-family>
  <skin-version>v1</skin-version>

</trinidad-config>
```

By default, JDeveloper configures the skyros skin family for a JSF application that uses ADF Faces. You can change this to fusion, 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 more information about creating custom skins, see Chapter 31, "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' : 'skyros'}) 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 Section A.6.2, "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. For more information, see Section A.8, "Using the RequestContext EL Implicit Object."

A.6.2 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.

A.6.2.1 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. For more information, see Chapter 31, "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`.

> **Note:** Enabling animation will have an impact on performance. For more information, see the "Oracle Application Development Framework Performance Tuning" chapter of *Tuning Performance*.

A.6.2.2 Skin Family

As described in Section A.6.1, "How to Configure ADF Faces Features in trinidad-config.xml," JDeveloper by default uses the `skyros` skin family for a JSF application that uses ADF Faces. You can change the `<skin-family>` value to `fusion`, `simple`, or to a custom skin definition. For information about creating and using custom skins, see Chapter 31, "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>
```

A.6.2.3 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`.

A.6.2.4 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.

A.6.2.5 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.
- `screenReader`: Accessibility-specific constructs are added to improve behavior under a screen reader.

> **Note:** Screen reader mode may have a negative effect on other users. For example, access keys are not displayed if the accessibility mode is set to screen reader mode.

Use `<accessibility-profile>` to configure the color contrast and font size used in the application. The supported values are:

- `high-contrast`: Application displays using high-contrast instead of the default contrast.
- `large-fonts`: Application displays using large fonts instead of the default size fonts.

To use more than one setting, separate the values with a space.

A.6.2.6 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>
```

A.6.2.7 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>
  ```

A.6.2.8 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.

A.6.2.9 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.

A.6.2.10 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.

A.6.2.11 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.

A.6.2.12 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>
```

A.6.2.13 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.

A.6.3 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
```

A.7 Configuration in trinidad-skins.xml

By default, JDeveloper uses the skyros 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 Section A.6.1, "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 detailed information about creating custom skins, see Chapter 31, "Customizing the Appearance Using Styles and Skins."

A.8 Using the RequestContext EL Implicit Object

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())
{
  .
  .
  .
}
```

A.9 Performance Tuning

In addition to the performance tips related to specific configuration options, find more information about performance tuning in the "Oracle Application Development Framework Performance Tuning" chapter of *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:

- Section B.1, "About ADF Faces Default Messages"
- Section B.2, "Message Keys and Setter Methods"
- Section B.3, "Converter and Validator Message Keys and Setter Methods"

B.1 About ADF Faces Default Messages

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 more information about overriding default converter and validator error messages globally, see Section 19.3.2, "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. For more information, see Section 19.3.2, "How to Define Custom Validator and Converter Messages for All Instances of a Component."

B.2 Message Keys and Setter Methods

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.

B.3 Converter and Validator Message Keys and Setter Methods

The following subsections give the reference details for all ADF Faces converter and validator detail messages.

B.3.1 af:convertColor

Converts strings representing color values to and from `java.awt.Color` objects. The set of patterns used for conversion can be overriden.

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

B.3.2 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

B.3.3 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

B.3.4 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

B.3.5 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

B.3.6 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

B.3.7 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

B.3.8 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

B.3.9 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:

- Section C.1, "About Keyboard Shortcuts"
- Section C.2, "Tab Traversal"
- Section C.3, "Shortcut Keys"
- Section C.4, "Default Cursor or Focus Placement"
- Section C.5, "The Enter Key"

C.1 About Keyboard Shortcuts

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.

C.2 Tab Traversal

Tab traversal allows the user to move the focus through different UI elements on a page.

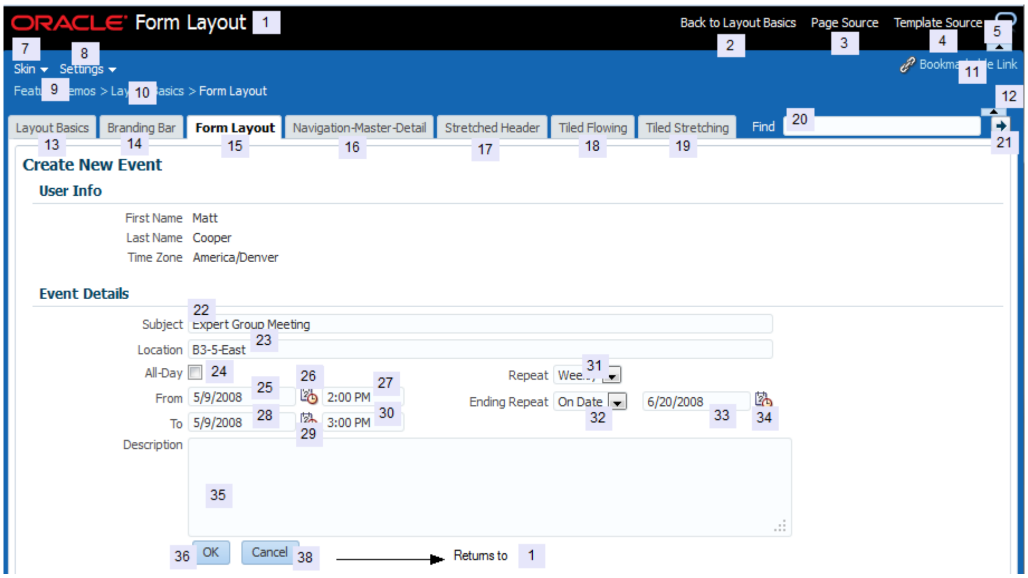
All active elements of the page are accessible by Tab traversal, that is, by using the Tab key to move to the next control and Shift+Tab to move to the previous control. In most cases, when a control has focus, the action can then be initiated by pressing Enter.

Some complex components use arrow keys to navigate after the component receives focus using the Tab key.

C.2.1 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.

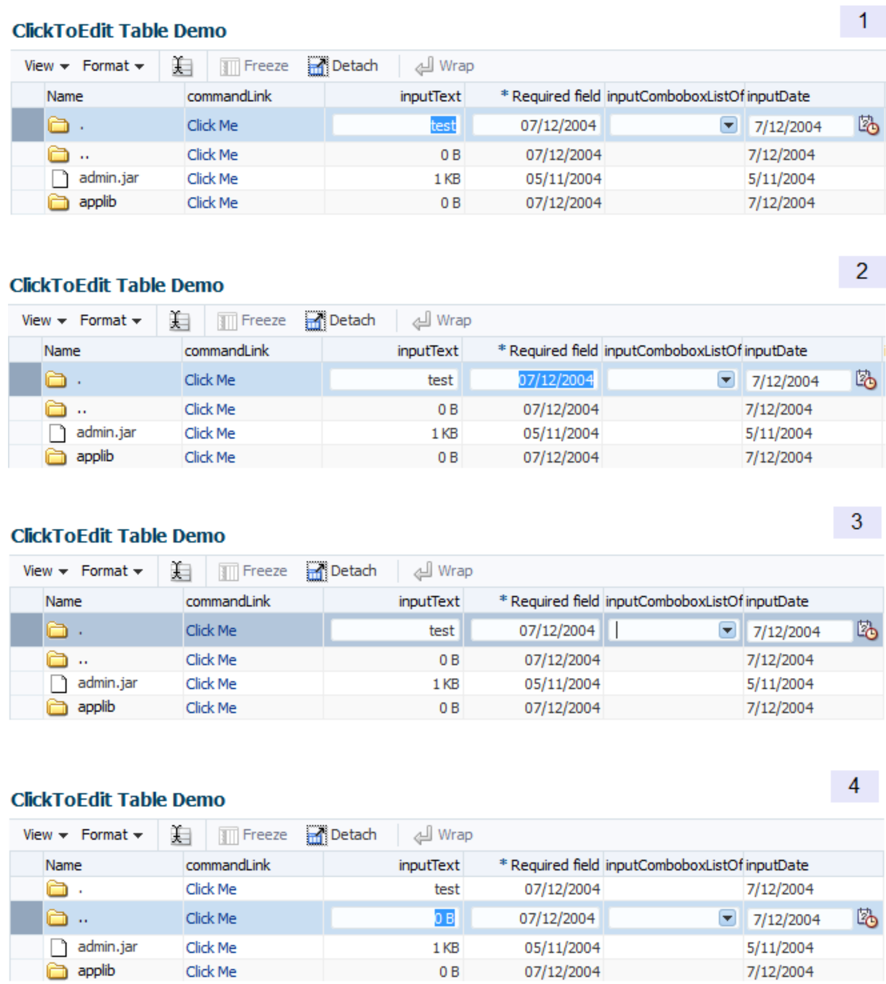
To improve keyboard navigation efficiency for users, you should set the `initialFocusId` attribute on the document. For accessibility purposes, you should also define a `skipLinkTarget` and include a `skip navigation` link at the top of the page, which should navigate directly to the first content-related tab stop.

C.2.2 Tab Traversal Sequence in a Table

The Tab traversals in a table establish a unique row-wise navigation pattern when the user presses the Tab key to navigate sequentially from one cell to another. When the user presses Enter, the focus moves to the next row, to follow the same pattern. The navigational sequence begins and ends in the same column as in the previous row.

Figure C–2 shows an example of a tab traversal sequence in a table.

Figure C–2 Tab Traversal Sequence in a Table



In Figure C–2, the user has navigated the rows 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.

   When the user presses the Enter key in an editabale field, the focus moves to the first editable field in the next row, and sets a navigation pattern based on the first set of Tab keys, which is followed in subsequent rows.

Note: The navigational pattern is not recognized if you use arrow keys to navigate from one cell to another.

C.3 Shortcut Keys

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.

C.3.1 Accelerator Keys

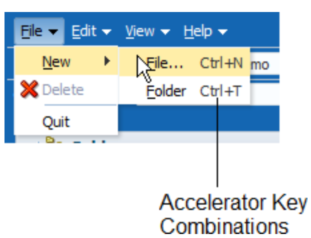
Accelerator keys bypass menu and page navigation and perform actions directly. Accelerator keys are sometimes also called *hot keys*. Common accelerator keys in a Windows application, such as Internet Explorer, are Ctrl+O for Open and Ctrl+P for Print.

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, and should be available in both the regular mode and screen reader mode.

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 Ctrl+Shift+W	Pop-up Messaging Secondary Windows	Toggle focus between open popups.
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–8 in menus when the related component also appears on the same page.

C.3.2 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

Notes:

- 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.

C.3.3 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.
Ctrl+Shift+Home Ctrl+Shift+End	Multi-Select Choice List Multi-Select List Box	Select all items from top to current selection, or select all items from current selection to bottom.
Arrow Left Arrow Right	Menu Bar Splitter Input Number Slider Input Range Slider Input Number Spinbox	Move focus to different menu on a menu bar. Move splitter left or right when it is in focus. 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 Arrow Down	Menu Splitter Input Number Slider Input Range Slider	Move focus to different menu items in a menu. Move splitter up or down when it is in focus. Move slider up or down when input number slider or input range slider is in focus.

C.3.4 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 Arrow Down/Arrow Up	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.
Ctrl+Alt+R	Active Data	If in screen reader mode, open the Active Data dialog. 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.
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.

C.3.5 Shortcut Keys for Screen Reader Mode

In screen reader mode, ADF Faces components may render differently than in the default mode in order to provide better accessibility. For example, a Tree component has an additional column of radio controls for tree node selection per row.

Figure C–5 and Figure C–6 show the ClickToEdit demo application in the default and the screen reader mode. In the screen reader mode, note the additional column of radio controls for row selection.

Figure C–5 ClicktoEdit Demo application in Default mode

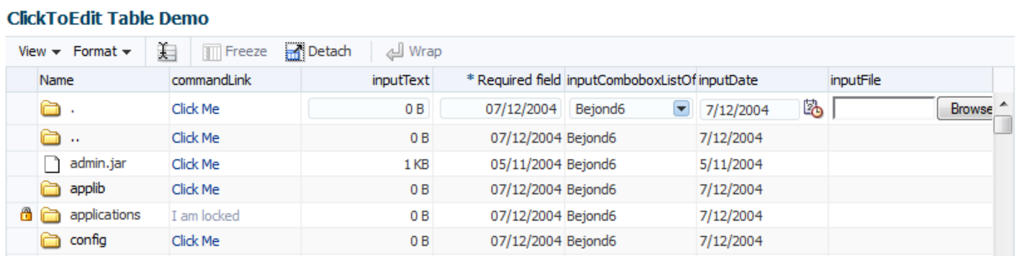
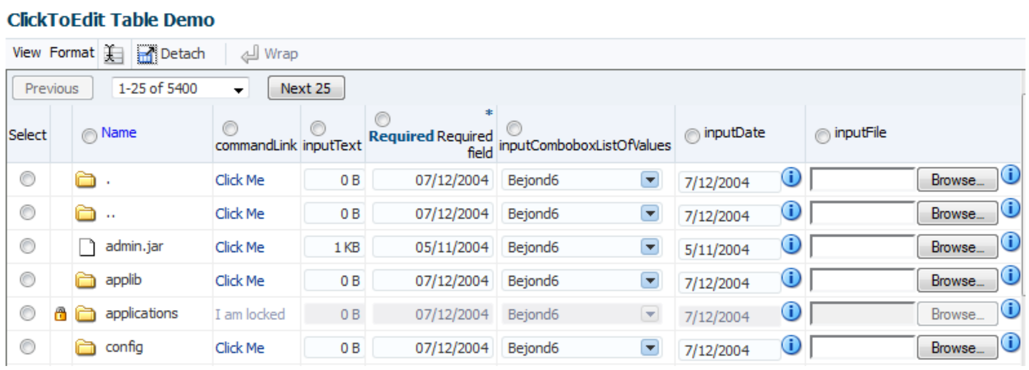


Figure C–6 ClickToEdit Demo application in Screen Reader mode



C.3.6 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

Table C–5 (Cont.) Shortcut Keys Assigned to Rich Text Editor Component

Shortcut Key	Components	Function
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
Ctrl+Y	Rich Text Editor	Redo
Ctrl+Z	Rich Text Editor	Undo

In screen reader mode, the Rich Text Editor component displays only the editor and the toolbar controls are not displayed. There are no shortcut keys for Rich Text Editor that apply in screen reader mode.

C.3.7 Shortcut Keys for Table, Tree, and Tree Table Components

Table C–6 lists shortcut keys assigned to Table, Tree, and Tree Table in regular rich mode. Shortcut keys for these components in screen reader mode is summarized in Table C–7. For more information about Tables and Trees, see Chapter 12, "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 Shift+Tab	Table Tree Table	Move focus to next or previous cell or editable component. 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.

Table C–6 (Cont.) Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Enter Shift+Enter	Table Tree Tree Table	Navigate to the next editable cell or previous editable cell of the column. 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. For more information, see Section C.2.2, "Tab Traversal Sequence in a Table."
Arrow Left Arrow Right	Table Tree Table	Move focus. In a table, when the focus is on an editable component, move the text cursor.
Arrow Up Arrow Down	Table Tree Table	Move focus. 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. In a table, when the focus is on an editable component that supports multiple options (such as `selectOneChoice` and `inputNumberSpinBox`), scroll the selected option. If the first row is selected, move focus to the column header. 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. **Note**: If `selectionEventDelay` is `enabled`, row selection during keyboard navigation is delayed by 300ms to allow table keyboard navigation without causing unwanted row selection.

Table C–6 (Cont.) Shortcut Keys Assigned to Table, Tree, and Tree Table components

Shortcut Key	Components	Function
Ctrl+Arrow Up Ctrl+Arrow Down	Table	Move focus. If in edit mode, submit the changes made in the current row and navigate to the previous row or next row. In the click-to-edit table, when the focus is on an editable component that supports multiple options (such as `selectOneChoice` and `inputNumberSpinBox`), scroll the selected option.
Ctrl+Arrow Left Ctrl+Arrow Right	Table	Move focus. If in edit mode, when the focus is on an editable component, move the text cursor.
Shift+Arrow Left Shift+Arrow Right	Table Tree Table	Move focus and add to selection.
Ctrl+Shift+Arrow Left Ctrl+Shift+Arrow Right	Table Tree Table	Move the selected column to the left or right.
Shift+Arrow Up Shift+Arrow Down	Table Tree Table Tree	Select multiple rows.
Page Up Page Down	Table Tree Table	If a row is selected, scroll and select the same row of the next or previous page. If no row is selected, scroll by one page.
Alt+Page Up Alt+Page Down	Table Tree Table	Horizontally scroll the table to the right or left.
Space Bar Ctrl+Space Bar	Table Tree Tree Table	Select the node. 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.

C.3.8 Shortcut Keys for Table, Tree, and Tree Table Components in Screen Reader Mode

Table C–7 lists shortcut keys assigned to the Tree, Table, and Tree Table components in screen reader mode.

Table C–7 Shortcut keys assigned to Components in Screen Reader mode

Shortcut Keys	Components	Function
Spacebar	Table Tree Tree Table	Select the component. To select a node, press Tab and move focus to the component or its radio control. Press Spacebar when the component is in focus. You can also use the Spacebar for multiple selection. Pressing Spacebar on a selected component, unselects it.
Enter	Table Tree	If the focus is on the column header, sort the column in ascending order. Pressing Enter again sorts the column in descending order. If the focus is on the disclosure icon of a tree, open or close that node. If the focus is on the filter cell, perform table filtering.
Tab Shift+Tab	Table	Move the focus to the next or previous cell or editable component.
Arrow Up Arrow Down Arrow Left Arrow Right	Table	Move focus.
Ctrl+Shift+Arrow Left Ctrl+Shift+Arrow Right	Table Tree Table	Move the selected column to the left or right.
Shift+Arrow Left Shift+Arrow Right	Table Tree Table	Resize the selected column by 25 pixels from the left or right.
Ctrl+Shift+^	Tree	If the focus is on the Hierarchical Selector icon, show the Hierarchical Selector popup.
Ctrl+Alt+M Ctrl+Alt+B	Table Tree Tree Table	Launch body context menu.

In screen reader mode, the Tree component has an additional column of radio controls for tree node selection per row. The next column in the tree contains the tree nodes themselves. Nodes that can be expanded have disclosure links that can be in either an open or closed state. Opening a closed disclosure link makes more nodes of the tree visible, and navigable on the page.

Note: The screen reader mode does not support the following functions for the Table and Tree Table components:

- Click-to-edit mode
- Select all rows from the first row to the current row or from the current row to the last row
- Select all rows
- Remove selection from all columns

C.3.9 Shortcut Keys for ADF Data Visualization Components

Table C–8 lists shortcut keys assigned to ADF Data Visualization Components including Gantt chart, ADF hierarchy viewer components, and ADF geographic map. For more information about ADF Data Visualization Components, see Chapter 22, "Introduction to ADF Data Visualization Components."

Table C–8 Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Arrow Left Arrow Right	List region of all Gantt chart types Chart region of project Gantt Chart region of scheduling Gantt Chart region of resource utilization Gantt ADF Geographic Map ADF Hierarchy Viewer - nodes Pivot table Pivot filter bar	Move focus. 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. 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 keys 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–8 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Arrow Up Arrow Down	List region of all Gantt chart types Chart region of project Gantt Chart Region of scheduling Gantt Chart region of resource utilization Gantt ADF Geographic Map ADF Hierarchy Viewer - nodes Pivot table Pivot filter bar	Move focus. If the focus is on the chart region of project Gantt, the arrow key navigation selects previous or next row. 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. 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. If the focus is on the ADF geographic map component, the arrow key navigation pans up or down by a small increment. 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. 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 the corresponding column header cell.
Page Up Page Down	ADF Geographic Map ADF Hierarchy Viewer - diagram	If the focus is on the ADF geographic map component, the page key navigation pans up or down by a large increment. If the focus is on the diagram of ADF hierarchy viewer, press and hold to Page Up or Page Down keys to pan up or down continuously. Press Ctrl+Page Up or Ctrl+Page Down to pan left or right continuously.
+	ADF Geographic Map ADF Hierarchy Viewer - diagram	Increase zoom level. If the focus is on the diagram of ADF 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.

Table C–8 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
-	ADF Geographic Map ADF Hierarchy Viewer - diagram	Decrease zoom level. If the focus is on the diagram of ADF 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.
Ctrl+Alt+M	All Gantt chart types Pivot table Pivot filer bar	Launch context menu.
Home	ADF Hierarchy Viewer - nodes	Move focus to first node in the current level.
End	ADF Hierarchy Viewer - nodes	Move focus to last node in the current level.
Ctrl + Home	ADF Hierarchy Viewer - nodes	Move focus and select the root node.
<	ADF Hierarchy Viewer - nodes	Switches to the active node's previous panel.
>	ADF Hierarchy Viewer - nodes	Switches to the active node's next panel.
Ctrl + /	ADF Hierarchy Viewer - nodes	Synchronize all nodes to display the active node's panel.
Ctrl+Shift+^	ADF Hierarchy Viewer - nodes	Go up one level.
Ctrl+/	ADF Hierarchy Viewer - nodes	Switch content panel.
Ctrl+Alt+0	ADF Hierarchy Viewer - diagrams	Center the active node and zoom the diagram to 100%.
Tab	ADF Hierarchy Viewer - nodes Pivot table Pivot filter bar	Move focus through elements.
Esc	ADF Hierarchy Viewer - nodes	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.
Spacebar	ADF 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	ADF Hierarchy Viewer - nodes Pivot table Pivot filter bar	Isolate and select active node. Press Shift+Enter to toggle the state of the node.
/	ADF Hierarchy Viewer - nodes	Toggle control panel state.

Table C–8 (Cont.) Shortcut Keys Assigned to ADF Data Visualization Components

Shortcut Key	Components	Function
Ctrl+F	ADF 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	ADF 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: ▪ 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.

In screen reader mode, the ADF Data Visualization Components are replaced by other accessible components, as described in Table C–9.

Table C–9 Alternate ADF Components for ADF Data Visualization Components in Screen Reader Mode

This ADF Data Visualization Component ...	... is replaced by
Chart region for each Gantt type	Hyperlinks Use the Tab key to move focus within the chart region. Use Enter or Esc keys to launch or dismiss the corresponding Properties dialog box.
ADF Geographic Map	Table Use standard keyboard shortcuts to navigate through the data, as described in Table C–6. Note that if the ADF geographic map instance references multiple data-bound themes, then a dropdown list is also rendered to enable end users to switch between the corresponding Table instances.
Hierarchy Viewer	Tree Table Use standard keyboard shortcuts to navigate through the data, as described in Table C–6.

In screen reader mode, the Pivot Table and Pivot Filter Bar render additional icons for each Header layer and Filter, respectively. End users of the screen reader software can use Tab key to move focus between these icons and press Enter to perform the specified Pivot operation.

C.3.10 Shortcut Keys for Calendar Component

The Calendar component has several views: Day view, Week view, Moth view, and List view. The List view is displayed in the screen reader mode.

Table C–10 lists shortcut keys assigned to the Calendar component.

Table C–10 Shortcut Keys Assigned to Calendar Component

Shortcut Key	Components	Function
Tab	Calendar	Move focus.
Shift+Tab		If the focus is on the calendar toolbar, move focus through Day, Week, Month, List, Forward button, Backward button, and Today button.
		In the day view, move focus through activities of the day.
		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.
		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.
		In the list view, move focus to the day, and then through the activities of the day.

Table C–10 (Cont.) Shortcut Keys Assigned to Calendar Component

Shortcut Key	Components	Function
Arrow Left Arrow Right	Calendar	Move focus. In the day view, Right and Left arrows do not move focus. 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. In the month view, the following interaction occurs: ■ If the focus is on a Month Day header label, move focus to the previous or next day label. 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. ■ 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. 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.

Table C–10 (Cont.) Shortcut Keys Assigned to Calendar Component

Shortcut Key	Components	Function
Arrow Up Arrow Down	Calendar	Move focus. 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. 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. In the month view, move focus through activities in a day. ■ 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	Launch context menu. You can also launch context menu by pressing Ctrl+Alt+B.

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.

C.3.11 Shortcut Keys for Calendar Component in Screen Reader Mode

In screen reader mode, the Calendar component renders to the List view. Table C–11 lists the shortcut keys assigned to Calendar component in screen reader mode.

Table C–11 Shortcut keys assigned to Calendar Component in Screen Reader mode

Shortcut Key	Components	Function
Tab Shift+Tab	Calendar	Move the focus to the next or previous cell or editable component.
Arrow Up Arrow Down Arrow Left Arrow Right	Calendar	Move focus.

C.4 Default Cursor or Focus Placement

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.

C.5 The Enter Key

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.
- In a table, the Enter key moves focus to the cell below, and pressing Shift+Enter moves focus to the cell above. When the focus moves, the current cell reverts to the read-only mode.

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:

- Section D.1, "Introduction to Creating Web Applications for Touch Devices Using ADF Faces"
- Section D.2, "How ADF Faces Behaves in Mobile Browsers on Touch Devices"
- Section D.3, "Best Practices When Using ADF Faces Components in a Mobile Browser"

D.1 Introduction to Creating Web Applications for Touch Devices Using ADF Faces

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.

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.

D.2 How ADF Faces Behaves in Mobile Browsers on Touch Devices

In touch devices, users touch the screen instead of clicking the mouse. The native browser then converts these touch events into mouse events for processing. In ADF Faces. component peers handle the conversion. 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 more information about peers, see Section 4.1, "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 (for more information about the touch event, see Table 6–3, " ADF Faces Client Events").

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)		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	Enabled	Zoom browser screen Zoom graph or map
Double-click to set anchor in the Hierarchy Viewer component	Double tap.	When the `setAnchorListener` 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.
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

Table D–1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
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.
	Single tap the Maximize icon	Maximizes the component	
	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.
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.

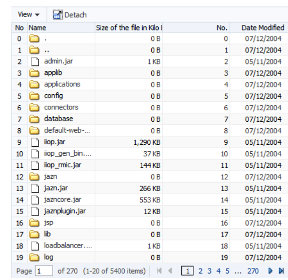
For further optimization, ADF Faces partitions JavaScript, so that the touch device JavaScript is separated from the desktop JavaScript. Only the needed JavaScript is downloaded when a page is rendered. Also, when a touch device is detected, CSS content specific to touch devices is sent to the page. 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
`selectManyShuttle` and `selectOrderShuttle`	Selection	Select boxes are displayed that allow users to select the item(s) to shuttle.

Table D–2 (Cont.) Component Differences in Mobile Browsers

Component	Functionality	Difference from desktop component
`table`	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.
`table`	Scroll	Instead of scroll bars, the table component is paginated, and displays a footer that allows the user to jump to specific pages of rows, as shown below.  The number of rows on a page is determined by the `fetchSize` attribute.
ADF Faces dialog framework	Windows	When a command component used to launch the dialog framework has its `windowEmbedStyle` attribute set to `window` (to launch in a separate window), ADF Faces overrides this value and sets it to `inlineDocument`, so that the dialog is instead launched inline within the parent window.
`menu`	Detachable menus	Detachable menus are not supported. The `detachable` attribute is ignored.
`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.
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.

D.3 Best Practices When Using ADF Faces Components in a Mobile Browser

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 Example D–1 to determine what device the application is being run on, and then deliver the correct page for the device.

Example D–1 Determining Platform

```
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	Set the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` web.xml parameter to `auto`. This setting ensures that the page will flow instead of stretch. For more information, see Section A.2.3.29, "Geometry Management for Layout and Table Components."
Partial page navigation	Using partial page navigation means that the JavaScript and other client code will not need to be downloaded from page to page, improving performance. For more information, see Section 8.5, "Using Partial Page Navigation."
Navigation	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 `panleSplitter` with a tree in the left pane to provide navigation, provide a list-based navigation model.
`document` tag	You can configure the `document` tag to reference a large icon (typically 129 pixels by 129 pixels). For more information, see Section 9.2.5, "How to Configure the document Tag."

Table D–3 (Cont.) Best Practices for ADF Faces Components in a Mobile Browser

Component/Functionality	Best Practice
Tables	By default, when rendered on tablet devices, tables display a footer that allows the user to jump to specific pages of rows. For all tables to display on a tablet device, you should: ■ Place the table components within a flowing container (that is, a component that does not stretch its children). ■ Set the `autoHeightRows` attribute to `0`. Better, is to set the `oracle.adf.view.rich.geometry.DEFAULT_DIMENSIONS` parameter to `auto`, as described for geometry management in the first row of this table. ■ Set the `scrollPolicy` attribute to `auto` (if the page may also run on a desktop device) or `page` (if the page will only run on a tablet. If the table is not in a flowing container, or if those attributes are not set correctly, the table will display a scroll bar instead of pages. For more information about table attributes, see Section 12.3.2, "Formatting Tables." For more information about flowing layouts and tables, see Section 12.2.7, "Geometry Management for the Table, Tree, and Tree Table Components."

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. 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. When you choose to apply a theme to the chosen quick layout, color and styling are added to some of the components used in the quick start layout.

Figure E–1 and Figure E–2 show each of the layouts with and without themes applied. For more information about themes, see Chapter 31, "Customizing the Appearance Using Styles and Skins"

Figure E–1 Quick Start Layouts With and Without Themes

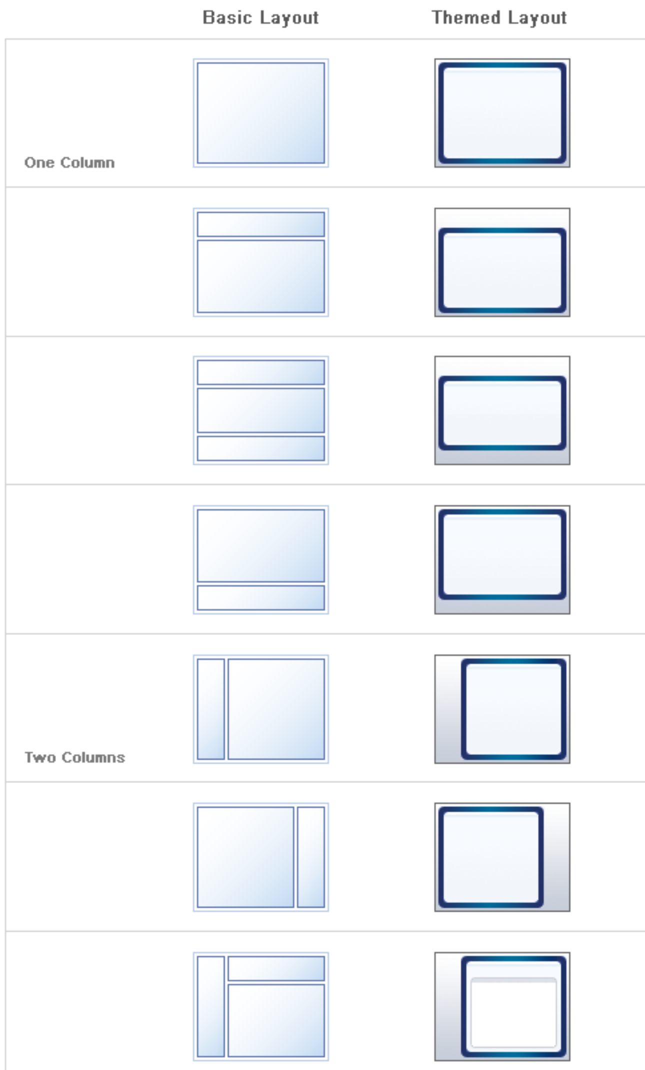


Figure E–2 Quick Start Layouts With and Without Themes

F Code Samples

This appendix provides the full length code samples referenced from sections throughout this guide.

This appendix includes the following sections:

- Section F.1, "Samples for Chapter 4, "Using ADF Faces Client-Side Architecture""
- Section F.2, "Sample for Chapter 23, "Using Graph Components""
- Section F.3, "Samples for Chapter 30, "Using Treemap and Sunburst Components""
- Section F.4, "Samples for Chapter 34, "Creating Custom ADF Faces Components""

F.1 Samples for Chapter 4, "Using ADF Faces Client-Side Architecture"

Following are code examples for using ADF Faces architecture

F.1.1 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. For more information, see Section 4.9, "JavaScript Library Partitioning." Example F–1 shows the default ADF Faces `adf-js-partitions.xml` file.

Example F–1 The Default 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>
```

F.2 Sample for Chapter 23, "Using Graph Components"

The following code example creates the `Employee` class, sample data, and drag and drop methods referenced in Section 23.7.6, "Adding Drag and Drop to Graphs."

Example F–2 dragAndDrop Managed Bean for Graph Drag and Drop Example

```
public class dragAndDrop {
  private Random m_random = new Random(23);
  // Simple class to represent employee data
  public static class Employee {
    public final String name;
    private int performance;
    private int potential;
    private double salary;
    private double bonus;
    private int experience;
    public Employee(String name, Random r) {
      this.name = name;
      this.performance = r.nextInt(100);
      this.potential = r.nextInt(100);
      this.salary = r.nextInt(100000);
      this.bonus = r.nextInt(50000);
      this.experience = r.nextInt(20);
    }
    // Returns a copy of the employee with a new name
    public Employee(String name, Employee e) {
      this.name = name;
      this.performance = e.performance;
      this.potential = e.potential;
      this.salary = e.salary;
      this.bonus = e.bonus;
      this.experience = e.experience;
    }
```

```
    public String getName() {return name;}
    public int getPerformance() { return performance; }
    public int getPotential() { return potential; }
    public void setSalary(double salary) { this.salary = salary; }
    public double getSalary() { return salary; }
    public void setBonus(double bonus) { this.bonus = bonus; }
    public double getBonus() { return bonus; }
    public int getExperience() { return experience; }
  }
  // Returns the employee represented by the selection
  private Employee findEmployee(List<Employee> list, GraphSelection selection) {
    if(!(selection instanceof DataSelection))
      return null;
    List<KeyMap> groupKeys = ((DataSelection)selection).getGroupKeys();
    for(KeyMap groupKey : groupKeys) {
      Set groupKeySet = groupKey.keySet();
      for(Object key : groupKeySet) {
        // Find the employee
        String name = groupKey.get((String)key);
        Employee emp = findEmployee(list, name);
        if(emp != null) {
          return emp;
        }
      }
    }
    return null;
  }
  private Employee findEmployee(List<Employee> list, String name) {
    for(Employee emp : list) {
      if(name.equals(emp.name))
        return emp;
    }
    return null;
  }
  // Converts the list of employees to a bubble data model
  private static DataModel createBubbleDataModel(List<Employee> employees) {
    Object[][] data = new Object[employees.size()*3][1];
    Object[] colLabels = new Object[employees.size()*3];
    Object[] rowLabels = {"Employees"};
    for(int i=0; i<employees.size(); i++) {
      Employee emp = employees.get(i);
      data[i*3][0] = emp.performance;
      data[i*3+1][0] = emp.salary;
      data[i*3+2][0] = emp.experience;
      colLabels[i*3] = emp.name;
      colLabels[i*3+1] = emp.name + "1";
      colLabels[i*3+2] = emp.name + "2";
    }
    LocalXMLDataSource dataSource = new LocalXMLDataSource(colLabels, rowLabels,
data);
    return new GraphDataModel(dataSource);
  }
  protected List<Employee> m_graphList = this.setEmployeeList();
  public List<Employee> setEmployeeList(){
        List<Employee> em_graphList = new ArrayList<Employee>();
        em_graphList.add(new Employee("Dan", m_random));
        em_graphList.add(new Employee("Ben", m_random));
        em_graphList.add(new Employee("Dave", m_random));
        em_graphList.add(new Employee("Chris", m_random));
        em_graphList.add(new Employee("Frank", m_random));
```

```
        em_graphList.add(new Employee("Jill", m_random));
        em_graphList.add(new Employee("Ray", m_random));
      return em_graphList;
  }
  public DataModel getGraphModel() {
    if(m_graphList == null) {
      setEmployeeList();
    }
    return createBubbleDataModel(m_graphList);
  }
  protected List<Employee> m_tableModel = this.setTableModelList();
  public List<Employee> setTableModelList(){
      List<Employee> em_tableModel = new ArrayList<Employee>();
      em_tableModel.add(new Employee("Brad", m_random));
      em_tableModel.add(new Employee("Derrick", m_random));
      em_tableModel.add(new Employee("Matt", m_random));
      return em_tableModel;
  }
  public List getTableModel() {
    if(m_tableModel == null) {
     setTableModelList();
    }
    return m_tableModel;
  }
  public DnDAction fromGraphDropListener(DropEvent event) {
    // Get the ComponentHandle from the transferable
    Transferable transferable = event.getTransferable();
    GraphSelectionSet selectionSet =
transferable.getData(GraphSelectionSet.class);
    // Now change each marker based on the DropEvent's proposed action
    DnDAction proposedAction = event.getProposedAction();
   for(GraphSelection selection : selectionSet) {
      Employee emp = findEmployee(m_graphList, selection);
      if(emp == null)
        return DnDAction.NONE;
      if(proposedAction == DnDAction.COPY) {
        m_tableModel.add(new Employee("Copy of " + emp.getName(), emp));
      }
      else if(proposedAction == DnDAction.LINK) {
        m_tableModel.add(new Employee("Link to " + emp.getName(), emp));
      }
      else if(proposedAction == DnDAction.MOVE) {
        m_graphList.remove(emp);
        m_tableModel.add(emp);
      }
    }
 RequestContext.getCurrentInstance().addPartialTarget(event.getDragComponent());
        return proposedAction;
  }
  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();
  }
```

F.3 Samples for Chapter 30, "Using Treemap and Sunburst Components"

Following are code examples for creating treemap and sunburst components.

F.3.1 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.

Example F–3 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.

Example F–3 Code Sample to Create a Treemap or Sunburst 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 Example F–3 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.

Example F–4 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.

Example F–4   Code Sample Creating Census Data Model for Treemap and Sunburst

```
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.

Example F–5 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.

Example F–5 Managed Bean Example to Set Census Data Treemap

```
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 Section F.3.2, "Code Sample for Sunburst Managed Bean" for an example.

F.3.2 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.

Example F–6 Managed Bean Example to Set Census Data Sunburst

```
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;
  }
}
```

F.4 Samples for Chapter 34, "Creating Custom ADF Faces Components"

Following are code examples for creating a custom component.

F.4.1 Event Code for JavaScript

When you create a custom component, you need to provide code in JavaScript that will perform the functions required when a event is fired, such as a mouse click. Example F–7 shows the event code that might be added for the `tagPane` component.

Example F–7 tagPane Event JavaScript

```
/**
 * Fires a select type event to the server for the source component
* when a tag is clicked.
*/
function AcmeTagSelectEvent(source, tag)
{
 AdfAssert.assertPrototype(source, AdfUIComponent);
 AdfAssert.assertString(tag); this.Init(source, tag);
}
// make AcmeTagSelectEvent a subclass of AdfComponentEvent

AdfObject.createSubclass(AcmeTagSelectEvent, AdfComponentEvent);
/**
 * The event type
*/
AcmeTagSelectEvent.SELECT_EVENT_TYPE = "tagSelect";
/**
 * Event Object constructor
*/
AcmeTagSelectEvent.prototype.Init = function(source, tag)
{
  AdfAssert.assertPrototype(source, AdfUIComponent);
  AdfAssert.assertString(tag);
  this._tag = tag;
 AcmeTagSelectEvent.superclass.Init.call(this, source, AcmeTagSelectEvent.SELECT_
EVENT_TYPE);}
/**
 * Indicates this event should be sent to the server
*/
AcmeTagSelectEvent.prototype.propagatesToServer = function()
{
  return true;
}
/**
 * Override of AddMarshalledProperties to add parameters * sent server side.
*/
AcmeTagSelectEvent.prototype.AddMarshalledProperties = function( properties)
{
  properties.tag = this._tag;


 }
/**
 * Convenient method for queue a AcmeTagSelectEvent.
 */
AcmeTagSelectEvent.queue = function(component, tag)
{
AdfAssert.assertPrototype(component, AdfUIComponent);
 AdfAssert.assertString(tag);
 AdfLogger.LOGGER.logMessage(AdfLogger.FINEST,
"AcmeTagSelectEvent.queue(component, tag)");
 new AcmeTagSelectEvent(component, tag).queue(true);
```

```
}
/**
 * returns the selected file type
*/
AcmeTagSelectEvent.prototype.getTag = function()
{
  return this._tag;}
/**
 * returns a debug string
*/
AcmeTagSelectEvent.prototype.toDebugString = function()
{
 var superString = AcmeTagSelectEvent.superclass.toDebugString.call(this);
 return superString.substring(0, superString.length - 1)
  +     ", tag="
  + this._tag     + "]";
}
/*
*
* Make sure that this event only invokes immediate validators
* on the client.
*/
AcmeTagSelectEvent.prototype.isImmediate = function()
{
  return true;
}
```

F.4.2 Example Tag Library Descriptor File Code

When you create a custom component, you need to create a tag library descriptor (TLD) file, which provides more information on the Java Class to the JSP compilation engine and IDE tools. Example F–8 shows an example TLD file that defines the `tagPane` component.

Example F–8 tagPane acme.tld Tag Library Descriptor Code

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<taglib 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-jsptaglibrary_2_1.xsd"
     version="2.1" xmlns="http://java.sun.com/xml/ns/javaee">
 <description>Acme Corporation JSF components</description>
 <display-name>acme</display-name>
 <tlib-version>1.0</tlib-version>
 <short-name>acme</short-name>
 <uri>http://oracle.adfdemo.acme</uri>
  <tag>
  <description>
  </description>
  <name>tagPane</name>
  <tag-class>oracle.adfdemo.acme.faces.taglib.TagPaneTag</tag-class>
  <body-content>JSP</body-content>
  <attribute>
   <name>id</name>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
   <name>rendered</name>
   <deferred-value>
    <type>boolean</type>
```

```
      </deferred-value>
    </attribute>
    <attribute>
      <name>tagSelectListener</name>
      <deferred-method>
        <method-signature>void
        </method-signature>
            myMethod(oracle.adfdemo.acme.faces.event.TagSelectEvent)
      </deferred-method>
    </attribute>
    <attribute>
      <name>visible</name>
      <deferred-value>
        <type>boolean</type>
      </deferred-value>
    </attribute>
    <attribute>
      <name>partialTriggers</name>
      <deferred-value>
      </deferred-value>
    </attribute>
    <attribute>
      <name>inlineStyle</name>
      <deferred-value/>
    </attribute>
    <attribute>
      <name>inlineClass</name>
      <deferred-value/>
    </attribute>
    <attribute>
      <name>tags</name>
      <deferred-value/>
    </attribute>
    <attribute>
      <name>binding</name>
      <deferred-value/>
    </attribute>
    <attribute>
      <name>orderBy</name>
      <deferred-value/>
    </attribute>
  </tag>
</taglib>
```

F.4.3 Example Component Class Code

When you create a custom component, you need to create a Java class that contains property information and accessors. Example F–9 shows the code used to create the example TagPane component.

Example F–9 TagPane Component Class

```
package oracle.adfdemo.acme.faces.component;

import java.util.Map;
import java.util.Set;

import javax.el.MethodExpression;
```

```
import javax.faces.event.AbortProcessingException;
import javax.faces.event.FacesEvent;

import oracle.adf.view.rich.event.ClientListenerSet;

import oracle.adfdemo.acme.faces.event.TagSelectEvent;
import oracle.adfdemo.acme.faces.event.TagSelectListener;

import org.apache.myfaces.trinidad.bean.FacesBean;
import org.apache.myfaces.trinidad.bean.PropertyKey;
import org.apache.myfaces.trinidad.component.UIXObject;

public class TagPane
  extends UIXObject
{

  /**
  * <p>Establishes a <code>Type</code> bean that contains JSF component property
  * information.  This static public final property <code>TYPE</code> shadows a
  * property with the same name in the super class.  The <code>TYPE</code>
  * attribute is defined once pre JSF component class.  The factory method call to
  * <code>FacesBean.Type</code> passes the super class reference
  * <code>UIXObject.TYPE</code> which copies down property information defined in
  * the super type.</p>
   */
  static public final FacesBean.Type TYPE =
    new FacesBean.Type(UIXObject.TYPE);

  /**
   * <p>Custom CSS applied to the style attribute of the root markup node.</p>
   */
  static public final PropertyKey INLINE_STYLE_KEY =
    TYPE.registerKey("inlineStyle", String.class);

  /**
   * <p>Custom CSS class to the class attribute of the root markup node.</p>
   */
  static public final PropertyKey STYLE_CLASS_KEY =
    TYPE.registerKey("styleClass", String.class);

  /**
   * <p>Key used to identify the <code>visible</code> property that
   * is of type <code>Boolean.class</code>.</p>
   */
  static public final PropertyKey VISIBLE_KEY =
    TYPE.registerKey("visible", Boolean.class, Boolean.TRUE);


  /**
   * <p>Key used to identify the <code>partialTriggers</code> property that
   * is of type <code>String[].class</code>.</p>
   */
  static public final PropertyKey PARTIAL_TRIGGERS_KEY =
    TYPE.registerKey("partialTriggers", String[].class);


  /**
   * <p>Key used to identify the <code>tagSelectListener</code> property that
   * is of type <code>MethodExpression.class</code>.</p>
   */
```

```
static public final PropertyKey TAG_SELECT_LISTENER_KEY =
  TYPE.registerKey("tagSelectListener", MethodExpression.class);

/**
 * <p>Key used to identity the <code>clientComponent</code> flag that
 * is of type <code>Boolean.class</code>.</p>
 */
static public final PropertyKey CLIENT_COMPONENT_KEY =
  TYPE.registerKey("clientComponent", Boolean.class);


/**
 * <p>Key use to identity the <code>clientListeners</code> property
 * that is a collection class of <code>ClientListenerSet.class</code>.</p>
 */
static public final PropertyKey CLIENT_LISTENERS_KEY =
  TYPE.registerKey("clientListeners", ClientListenerSet.class,
                   PropertyKey.CAP_NOT_BOUND);

/**
 * <p>Key used to identify the <code>clientAttributes</code> property this
 * is a set of type <code>PropertyKey.CAP_NOT_BOUND</code>.</p>
 */
static public final PropertyKey CLIENT_ATTRIBUTES_KEY =
  TYPE.registerKey("clientAttributes", Set.class,
                   PropertyKey.CAP_NOT_BOUND);

/**
 * <p>Key used to identify the <code>tags</code> property that
 * is of type <code>Map.class</code>.</p>
 */
static public final PropertyKey TAGS_KEY =
  TYPE.registerKey("tags", Map.class);


/**
 * <p>Key used to identify the <code>orderBy</code> property that
 * is of type <code>String.class</code>.</p>
 */
static public final PropertyKey ORDER_BY_KEY =
  TYPE.registerKey("orderBy", String.class);

/**
 * <p>The constructor calls the super classes overloaded constructor passing
 * the <code>rendererType</code>.
 */
public TagPane()
{
  // Pass the renderer type to the super class.
  super(RENDERER_TYPE);

  // requires a client component by default
  setBooleanProperty(CLIENT_COMPONENT_KEY, Boolean.TRUE);
}


/**
 * @param newpartialTriggers array of render dependent client ids
 */
public void setPartialTriggers(String[] newpartialTriggers)
```

```
  {
    setProperty(PARTIAL_TRIGGERS_KEY, newpartialTriggers);
  }

  /**
   * @return array of render dependent client ids
   */
  public String[] getPartialTriggers()
  {
    return (String[]) getProperty(PARTIAL_TRIGGERS_KEY);
  }


  /**
   * <p>Sets client listeners that provide specialized behaviors such as
   * drag-and-drop.
   * You can subscribe to any client event using the event type and method.
   * For example, we could add a client listener for our custom client event
   * <code>oracle.adfdemo.acme.js.event.TagSelectEvent</code>.
   * </p>
   * <pre>
   *     &lt;acme:tagPane tags="#{bean.tags}"&gt;
   *         &lt;clientListener type="tagSelect" method="onClickType" /&gt;
   *     &lt;\acme:tagPane&gt;
   * </pre>
   *
   * <p>In the fragment above, the method "onClickType" is a JavaScript function
that
   * has a single event parameter.</p>
   *
   * <per>
   *    function onClickType(event) {
   *       alter("You clicked on tag: " + event.getTag());
   *    }
   * </per>
   *
   * @param newclientListeners  client listener collection
   */
  public void setClientListeners(ClientListenerSet newclientListeners)
  {
    setProperty(CLIENT_LISTENERS_KEY, newclientListeners);
  }

  /**
   * <p>Returns client listeners that provide specialized behaviors such as
   * drag-and-drop.</p>
   *
   * @return client listener set
   */
  public ClientListenerSet getClientListeners()
  {
    return (ClientListenerSet) getProperty(CLIENT_LISTENERS_KEY);
  }

  /**
   * <p>Sets attributes on the client component,
   * <code>oracle.adfdemo.acme.js.event.AcmeTagPane.js</code> that can be used by
   * <code>ClientListners</code>.  The state is actually managed by the
   * component.  This collection is a <code>Set</code> of client attribute names
   * that are also server component properties.
```

```
   * <br/><br/>
   * Client attributes must be set using the <code>setClientAttribute</code> JSP
tag.
   * Consider the following:
   * <pre>
   *     &lt;acme:tagPane targetFolder="somefolder"&gt;
   *         &lt;clientListener type="tagSelect" method="onClickType" /&gt;
   *         &lt;clientAttribute name="whoareyou" method="While E. Coyote" /&gt;
   *     &lt;\acme:tagPane&gt;
   * </pre>
   *
   * <p>In the fragment above, the method "onClickType" is a JavaScript function
that
   * has a single event parameter.</p>
   *
   * <per>
   *    function onClickType(event) {
   *       alter(event.getSource().getProperty("whoareyou")
   *           + " clicked on tag: " + event.getTag());
   *    }
   * </per>
   *
   * @param newclientAttributes a set of client attribute names
   */
  public void setClientAttributes(Set newclientAttributes)
  {
    setProperty(CLIENT_ATTRIBUTES_KEY, newclientAttributes);
  }

  /**
   * <p><code>Set</code> of client component attribute names.  The state
   * is also kept in the server side component's <code>attributes</code> Map.</p>
   *
   * @return colleciton of unique client component attribute names
   */
  public Set getClientAttributes()
  {
    return (Set) getProperty(CLIENT_ATTRIBUTES_KEY);
  }

  /**
   * <p>Sets the <code>newselectListener</code> method binding
   * expression that expects a single parameter of type {@link
   * oracle.adfdemo.acme.faces.event.TagSelectEvent}.
   * This binding will be when the client-side
   * <code>oracle.adfdemo.acme.js.event.AcmeTagSelectEvent.js</code>
   * is queued from clicking on one of the <code>tag</code>s.</p>
   *
   * @param newselectListener invokes a {@link
oracle.adfdemo.acme.faces.event.TagSelectEvent} method expression
   */
  public void setTagSelectListener(MethodExpression newselectListener)
  {
    setProperty(TAG_SELECT_LISTENER_KEY, newselectListener);
  }

  /**
   * <p>Returns a method binding expression that is fired when the client-side
   * <code>oracle.adfdemo.acme.js.event.AcmeTagSelectEvent.js</code> is
   * queued.</p>
```

```
 *
 * @return invokes a {@link oracle.adfdemo.acme.faces.event.TagSelectEvent}
 * method expression
 */
public MethodExpression getTagSelectListener()
{
  return (MethodExpression) getProperty(TAG_SELECT_LISTENER_KEY);
}

/**
 * <p>Method for adding a listener implementing {@link
 * oracle.adfdemo.acme.faces.event.TagSelectListener}.</p>
 *
 * @param listener tag select listener
 */
public void addTagSelectListener(TagSelectListener listener)
{
  addFacesListener(listener);
}

/**
 * <p>Removes a {@link oracle.adfdemo.acme.faces.event.TagSelectListener}
 * listener.</p>
 *
 * @param listener tag select listener
 */
final public void removeSelectListener(TagSelectListener listener)
{
  removeFacesListener(listener);
}

/**
 * <p>Returns an array of attached {@link
 * oracle.adfdemo.acme.faces.event.TagSelectListener} listeners.</p>
 *
 * @return  an array of attached select listeners.
 */
final public TagSelectListener[] getSelectListeners()
{
  return (TagSelectListener[]) getFacesListeners(TagSelectListener.class);
}


/**
 * <p>Exposes the <code>FacesBean.Type</code> for this class through a protected
 * method.  This method is called but the <code>UIComponentBase</code> super
 * class to setup the components <code>ValueMap</code> which is the container
 * for the <code>attributes</code> colleciton.</p>
 *
 * @return <code>FolderSummary.TYPE</code> static property
 */
@Override
protected FacesBean.Type getBeanType()
{
  return TYPE;
}

/**
 * <p>
 * </p>
```

```
 *
 * @param facesEvent faces event
 * @throws AbortProcessingException exception during processing
 */
@Override
public void broadcast(FacesEvent facesEvent)
  throws AbortProcessingException
{

  // notify the bound TagSelectListener
  if (facesEvent instanceof TagSelectEvent)
  {
    TagSelectEvent event = (TagSelectEvent) facesEvent;

    // utility method found in UIXComponentBase for invoking method event
    // expressions
    broadcastToMethodExpression(event, getTagSelectListener());
  }

  super.broadcast(facesEvent);
}

/**
 * <p>CSS value applied to the root component's style attribute.</p>
 *
 * @param newinlineStyle CSS custom style text
 */
public void setInlineStyle(String newinlineStyle)
{
  // inlineStyle = newinlineStyle;
  setProperty(INLINE_STYLE_KEY, newinlineStyle);
}

/**
 * <p>CSS value applied to the root component's style attribute.</p>
 *
 * @return newinlineStyle CSS custom style text
 */
public String getInlineStyle()
{
  // return inlineStyle;
  return (String) getProperty(INLINE_STYLE_KEY);
}

/**
 * <p>CSS style class added to the components class attribute.</p>
 * @param newstyleClass custom class style attribute
 */
public void setStyleClass(String newstyleClass)
{
  setProperty(STYLE_CLASS_KEY, newstyleClass);
}

/**
 * <p>CSS style class added to the components class attribute.</p>
 * @return newstyleClass custom class style attribute
 */
public String getStyleClass()
{
  return (String) getProperty(STYLE_CLASS_KEY);
```

```
}

/**
 * <p>Sets the visibility of the component.  The visibility
 * is not the same as <code>rendered</code>.  The <code>visible</code>
 * attribute effects the CSS style on the CSS root of the component.
 * </p>
 *
 * @param newvisible <code>true</code> if the markup is not hidden in the
 * browser
 */
public void setVisible(boolean newvisible)
{
  setBooleanProperty(VISIBLE_KEY, newvisible);
}

/**
 * <p>Returns the visibility of the component.  The visibility
 * is not the same as <code>rendered</code>.  The <code>visible</code>
 * attribute effects the CSS style on the CSS root of the component.
 * </p>
 *
 * @return <code>true</code> if the markup is not hidden in the browser
 */
public boolean isVisible()
{
  return getBooleanProperty(VISIBLE_KEY, Boolean.TRUE);
}

/**
 * <p>This component's type, <code>oracle.adfdemo.acme.TagPane</code></p>
 */
static public final String COMPONENT_TYPE =
  "oracle.adfdemo.acme.TagPane";

/**
 * <p>Logical name given to the registered renderer for this component.</p>
 */
static public final String RENDERER_TYPE = "oracle.adfdemo.acme.TagPane";

/**
 * <p>Sets a Map of weighted tags.  The key represents the tag name
 * and the value a number.</p>
 *
 * @param newtags tags to be rendered
 */
public void setTags(Map<String, Number> newtags)
{
  setProperty(TAGS_KEY, newtags);
}

/**
 * <p>Gets a Map of weighted tags.  The key represents the tag name
 * and the value a number.</p>
 *
 * @return newtags tags to be rendered
 */
public Map<String, Number> getTags()
{
  return (Map<String, Number>) getProperty(TAGS_KEY);
```

```
  }

  /**
   * <p>Sets the order that the tags are rendered.
   * The valid enumerations are "alpha" and "weight".</p>
   *
   * @param neworderBy order by enumeration
   */
  public void setOrderBy(String neworderBy)
  {
    setProperty(ORDER_BY_KEY, neworderBy);
  }

  /**
   * <p>Gets the order that the tags are rendered.
   * The valid enumerations are "alpha" and "weight".</p>
   *
   * @return order by enumeration
   */
  public String getOrderBy()
  {
    return (String) getProperty(ORDER_BY_KEY);
  }


   static {
    // register the new TYPE by family and rendererType
     TYPE.lockAndRegister(UIXObject.COMPONENT_FAMILY, RENDERER_TYPE);
   }

}
```

F.4.4 Example Renderer Class Code

The renderer class controls the display of a component. Example F–10 shows the renderer class for the custom TagPane component.

Example F–10 Renderer Class

```
package oracle.adfdemo.acme.faces.render;

import java.io.IOException;

import java.text.MessageFormat;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.logging.Logger;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

import oracle.adf.view.rich.render.ClientComponent;
import oracle.adf.view.rich.render.ClientEvent;
import oracle.adf.view.rich.render.ClientMetadata;
```

```
import oracle.adf.view.rich.render.RichRenderer;

import oracle.adfdemo.acme.faces.component.TagPane;
import oracle.adfdemo.acme.faces.event.TagSelectEvent;

import org.apache.myfaces.trinidad.bean.FacesBean;
import org.apache.myfaces.trinidad.bean.PropertyKey;
import org.apache.myfaces.trinidad.context.RenderingContext;


/**
 * <p>Renderer for {@link oracle.adfdemo.acme.faces.component.TagPane}
 * component.</p>
 */
public class TagPaneRenderer
  extends RichRenderer
{

  /**
   * <p>Java Logger instance.</p>
   */
  private static final Logger _LOG =
    Logger.getLogger(TagPaneRenderer.class.getName());


  /**
   * <p>No-args constructor passes the <code>TagPane.TYPE</code> to the
   * base class.  The type bean is used to extend the component properties
   * meta-data by invoking <code>findTypeConstants</code>.</p>
   */
  public TagPaneRenderer()
  {
    super(TagPane.TYPE);
  }


  /**
   * <p>Invoked by the base class, this is the hook to perform the markup
   * rendering for the {@link oracle.adfdemo.acme.faces.component.TagPane}.
   * </p>
   *
   * @param context faces context
   * @param arc rendering context
   * @param component {@link oracle.adfdemo.acme.faces.component.TagPane}
   * @param client server-side state for the client component
   * @param bean state holder for the TagPane component
   * @throws IOException error rendering markup
   */
  protected void encodeAll(FacesContext context, RenderingContext arc,
                           UIComponent component, ClientComponent client,
                           FacesBean bean)
    throws IOException
  {

    //defines this attribute to be pushed to the client component
    //this attribute is not secured so its value will prorogate
    //to the server when modified from the client
    client.addProperty(_ORDER_BY_KEY, bean.getProperty(_ORDER_BY_KEY));

    ResponseWriter writer = context.getResponseWriter();
```

```
    writer.startElement("div", component);

    // write the client id for the component
    renderId(context, component);

    // add the component root style attribute; factors in the inline and class
    // style component properties
    renderRootStyleAttributes(context, arc, client, bean,
                              getDefaultStyleClass(context, arc, bean));

    writer.startElement("span", component);

    // writes the style class attribute using a skinning key;  the key
    // will need to be normalized and compressed
    renderStyleClass(context, arc, TagPaneRenderer._TAG_CONTENT_STYLE_KEY);

    _renderTags(context, component, arc, bean);

    writer.endElement("span");

    writer.endElement("div");

  }

  /**
   * <p>Renders the <code>tags</code> as anchored links ordered by
   * tag or tag weight.  The tag weight is converted into a font that is
   * evenly distributed between the <code>MAXIMUM_FONT_SIZE</code> and the
   * <code>MINIMUM_FONT_SIZE</code> based on the tag's weight.</p>
   *
   * @param context faces context
   * @param component {@link oracle.adfdemo.acme.faces.component.TagPane}
   * @param arc rendering context
   * @param bean state holder for {@link
oracle.adfdemo.acme.faces.component.TagPane}
   * @throws IOException error during rendering tags
   */
  private void _renderTags(FacesContext context, UIComponent component,
                           RenderingContext arc, FacesBean bean)
    throws IOException
  {
    Map<String, Number> tags = _getTags(bean);
    if (tags.isEmpty())
    {
      return; // no work to do
    }

    // Calculates a font size for each tag.  The font size will be based on the
    // weight of the tag and the size proportional to the range of font sizes.
    Map<String, Integer> fontSizes =
      _computeFontSize(tags, _getMaximumFontSize(arc),
                       _getMinimumFontSize(arc));

    // create a list from the map's entry set.
    List<Map.Entry<String, Number>> entries =
      new ArrayList<Map.Entry<String, Number>>(tags.entrySet());

    //collocation sequence determined by the orderBy property
    Collections.sort(entries, _getComparator(bean));
```

```
  ResponseWriter writer = context.getResponseWriter();

  //for each tag, write an anchored tag.  The font size is explicitly set
  //using a inline style.
  for (Map.Entry<String, Number> entry: entries)
  {
    // start a hyperlink for each tag
    writer.startElement("a", component);

    // render the class style for the link
    renderStyleClass(context, arc, TagPaneRenderer._TAG_STYLE_KEY);

    writer.writeAttribute("href", "#", null);

    //font size is calculated from the _computeFontSize call above
    //the _getFontSizeStyle substitutes the font size parameter into the style
    String style = _getFontSizeStyle(fontSizes.get(entry.getKey()));

    //set the font size using an inline style
    writer.writeAttribute("style", style, null);

    //insert the tag weight into the title string
    String title = _getTitleStyle(entry.getValue().doubleValue(), arc);

    // write the title string attribute to the hyperlink
    writer.writeAttribute("title", title, null);

    //write body of the tag; the description of the link will
    //be the name of the tag
    writer.writeText(entry.getKey(), null);

    //end the hyperlink
    writer.endElement("a");

    // write a small spacer between anchored tags
    renderSpacer(context, arc, "2px", "2px");
  }

}

/**
 * <p>Substitutes the tag's <code>weight</code> into the pattern defined by
 * the <code>AcmeTagPane_tag_title</code> resource key.</p>
 * @param weight tag weight
 * @param arc rendering context
 * @return formatted title attribute for the anchored tag
 */
private String _getTitleStyle(double weight, RenderingContext arc)
{
  // title attribute template from the resource bundle
  MessageFormat titleFormat =
    new MessageFormat(arc.getTranslatedString("AcmeTagPane_tag_title"));

  return titleFormat.format(new Object[]
      { weight });
}

/**
 * <p>Substitutes the <code>size</code> into the style pattern defined by
```

```
 * the <code>_FONT_INLINESTYLE_FORMAT</code>.</p>
 * @param size font size
 * @return inline style defining the font size
 */
private String _getFontSizeStyle(int size)
{
  return _FONT_INLINESTYLE_FORMAT.format(new Object[]
      { size });
}

/**
 * <p>Returns the default constructor for our client-side component,
 * <code>oracle.adfdemo.acme.js.component.AcmeTagPane.js</code></p>
 *
 * @return client-side component name
 */
protected String getClientConstructor()
{
  return _CLIENT_COMPONENT;
}


/**
 * <p>Retrieves a hint on if and when a component must be sent down to the
 * client.</p>
 *
 * @return the client component type
 */
@Override
protected ClientComponent.Type getDefaultClientComponentType()
{
  return ClientComponent.Type.CREATE_WITH_REQUIRED_ATTRS;
}

/**
 * <p>Retrieves the style class to apply to the root element,
 * <code>_ROOT_STYLE_KEY</code>.</p>
 *
 * @param context the FacesContext
 * @param arc the RenderingContext
 * @param bean the FacesBean of the component to render
 * @return the root element's style class
 */
@Override
protected String getDefaultStyleClass(FacesContext context,
                                      RenderingContext arc,
                                      FacesBean bean)
{
  return TagPaneRenderer._ROOT_STYLE_KEY;
}


/**
 * <p>Returns the weighted tag map from the component's state holder
 * properties bag using the <code>_TAGS_KEY</code>.</p>
 *
 * @param bean state holder for {@link
 * oracle.adfdemo.acme.faces.component.TagPane}
 * @return map of weighted tags
 */
```

```
private Map<String, Number> _getTags(FacesBean bean)
{
  Map<String, Number> tags =
    (Map<String, Number>) bean.getProperty(_TAGS_KEY);
  if (tags == null)
  {
    tags = Collections.emptyMap();
  }

  return tags;
}

/**
 * <p>Looks at the {@link oracle.adfdemo.acme.faces.component.TagPane}'s
 * <code>orderBy</code> property and returns a corresponding
 * <code>Comparator</code>.  The default is to sort descending by weight.
 * If <code>orderBy</code> equals "alpha", the order is sorted ascending
 * by the tag.</p>
 *
 * @param bean state holder for the {@link
 *oracle.adfdemo.acme.faces.component.TagPane}
 * @return comparator to sort the <code>tags</code>
 */
private Comparator<Map.Entry<String, Number>> _getComparator(FacesBean bean)
{
  String orderBy = (String) bean.getProperty(_ORDER_BY_KEY);
  if (orderBy != null && orderBy.equalsIgnoreCase("alpha"))
  {
    return _sortAscByTag;
  }

  return this._sortDescByWeight;
}

/**
 * <p>This method is called from the <code>RichRenderer</code>'s implementation
 * of the <code>decode</code> method.  Within this method we will check to see
 * if there are any custom client events,
 * <code>oracle.adfdemo.acme.js.event.AcmeTagSelectEvent</code>.
 * If a client event is present, we will queue a corresponding server-side faces
 * event, {@link oracle.adfdemo.acme.faces.event.TagSelectEvent}.  This event is
 * targeted for the apply request values phase so the response will be completed
 *  if a client event is found.
 * </p>
 *
 * @param facesContext faces context
 * @param component {@link oracle.adfdemo.acme.faces.component.TagPane}
 * component
 * @param clientId unique component identifier
 */
@Override
public void decodeInternal(FacesContext facesContext,
                           UIComponent component, String clientId)
{

  super.decodeInternal(facesContext, component, clientId);

  // look for a client event of type "tagSelect"
  ClientEvent clientEvent =
    this.getClientEvent(facesContext, clientId, _CLIENT_SELECT_EVENT_TYPE);
```

```
    if (clientEvent != null)
    {
      // extract the tag from the client event
      String tag =
        (String) clientEvent.getParameters().get(TagPaneRenderer._CLIENT_SELECT_
EVENT_TAG_PARAM);

      _LOG.finest("Found SelectEvent with tag of " +
                  (tag != null? tag: "null"));

      // Instantiate a corresponding server side event
      TagSelectEvent event = new TagSelectEvent(component, tag);

      // push the event up to the view root
      event.queue();

      // this event is an assumed immediate meaning that we will stop the
      //lifecycle after
      // the apply request values phase and short-circuit to render response.
      // there doesn't seem to be a good reason for this component to participate
      // in the process validations phase.  that is why we are going to jump to
      // the render response phase.
      facesContext.renderResponse();
    }

  }

  /**
   * <p>This method is called from the base constructor.  The callback gives
   * opportunity to add client characteristics to the server-side component
   * properties that become  client-side component properties.</p>
   *
   * @param type <code>FacesBean.TYPE</code>
   * @param metadata client specific extension to the <code>FacesBean.Type</code>
   */
  @Override
  protected void findTypeConstants(FacesBean.Type type,
                                  ClientMetadata metadata)
  {
    super.findTypeConstants(type, metadata);

    _TAGS_KEY = type.findKey("tags");
    metadata.addSecureProperty(_TAGS_KEY);

    _ORDER_BY_KEY = type.findKey("orderBy");

  }

  /**
   * <p>Calculates a proportional font weight for each of the <code>tags</code>
   * based on the weight of the tag entry.  The weight of each entry is
   * determined by the value of the <code>Map.Entry</code>.  The computed font
   * weight for each item will be between the <code>maxFontSize</code> and
   * <code>minFontSize</code>. Tags that have the same weight will be
   * displayed using the same font.</p>
   *
   * @param tags weighted tags
   * @param maxFontSize maximum font weight
   * @param minFontSize minimum font weight
   * @return Lookup map that contains the tag name as the key and the value
```

```
 * is the computed font size.
 */
private Map<String, Integer> _computeFontSize(Map<String, Number> tags,
                                              double maxFontSize,
                                              double minFontSize)
{

  double maxWeight = 0; // max weight
  double minWeight = 0; // min weight

  // find the min and max in the tag weight set
  for (Map.Entry<String, Number> entry: tags.entrySet())
  {
    // first time set the min to the first entry
    if (minWeight == 0)
    {
      minWeight = entry.getValue().doubleValue();
    }

    maxWeight = Math.max(maxWeight, entry.getValue().doubleValue());
    minWeight = Math.min(minWeight, entry.getValue().doubleValue());
  }

  //target map where the key is the tag and the value the font size
  Map<String, Integer> fontSizeMap = new TreeMap<String, Integer>();


  // for each entry, calculate a font size adding to the target
  // map by tag name
  for (Map.Entry<String, Number> entry: tags.entrySet())
  {

    double weight = entry.getValue().doubleValue();
    double percentTotaWeight =
      (weight - (minWeight - 1d)) / ((maxWeight + 1d));
    double fontSize =
      (percentTotaWeight * ((maxFontSize + 1d) - minFontSize)) +
      (minFontSize - 1d);

    //add to the xref map
    fontSizeMap.put(entry.getKey(), (int) Math.round(fontSize));

  }

  return fontSizeMap;
}

/**
 * <p>Returns the minimum font size that will be used as the lower limit when
 * assigning each unique tag weight a font size.  The value will be pulled from
 * the skinning key "acme|tagPane-tr-minimum-font-size".  If the
 * key is not found or the value is not an integer, the
 * <code>_DEFAULT_MINIMUM_FONT_SIZE</code> will be used.</p>
 *
 * @param arc Rendering context
 * @return max font size assigned to a tag
 */
private int _getMinimumFontSize(RenderingContext arc)
{
  int size = _DEFAULT_MINIMUM_FONT_SIZE;
```

```
  String fontSize = (String)
    arc.getSkin().getProperty(_MINIMUM_FONT_SIZE_PROPERTY);
  if (fontSize != null)
  {
    try
    {
      size = Integer.parseInt(fontSize);
    }
    catch (NumberFormatException e)
    {
      _LOG.warning("The \"" + _MINIMUM_FONT_SIZE_PROPERTY +
                   "\" skin property " +
                   "is not defined or is not a valid integer. " +
                   "Using default minimum font size: " + size);
    }
  }
  return size;
}

/**
 * <p>Returns the maximum font size that will be used as the upper limit when
 * assigning each unique tag weight a font size.  The value will be pulled from
 * the skinning key "acme|tagPane-tr-maximum-font-size".  If the
 * key is not found or the value is not an integer, the
 * <code>_DEFAULT_MAXIMUM_FONT_SIZE</code> will be used.</p>
 *
 * @param arc Rendering context
 * @return max font size assigned to a tag
 */
private int _getMaximumFontSize(RenderingContext arc)
{
  int size = _DEFAULT_MAXIMUM_FONT_SIZE;
  String fontSize = (String)
    arc.getSkin().getProperty(_MAXIMUM_FONT_SIZE_PROPERTY);
  if (fontSize != null)
  {
    try
    {
      size = Integer.parseInt(fontSize);
    }
    catch (NumberFormatException e)
    {
      _LOG.warning("The \"" + _MAXIMUM_FONT_SIZE_PROPERTY +
                 "\" skin property " +
                 "is not defined or is not a valid integer. " +
                 "Using default maximum font size: " + size);
    }
  }
  return size;
}


/**
 * <p>Sorts the <code>tags.entrySet</code> by map value in descending order
 * and by map key in ascending.
 * </p>
 */
private Comparator<Map.Entry<String, Number>> _sortDescByWeight =
  new Comparator<Map.Entry<String, Number>>()
{
```

```
  public int compare(Map.Entry<String, Number> o1,
                     Map.Entry<String, Number> o2)
  {
    double value1 = o1.getValue().doubleValue();
    double value2 = o2.getValue().doubleValue();
    if (value1 > value2)
    {
      return -1;
    }
    else if (value1 < value2)
    {
      return 1;
    }
    else
    {
      // sort by value DESC, key ASC
      return o1.getKey().compareTo(o2.getKey());
    }
  }
};


/**
 * <p>Sorts the <code>tags.entrySet</code> by the key in ascending order.
 * </p>
 */
private Comparator<Map.Entry<String, Number>> _sortAscByTag =
  new Comparator<Map.Entry<String, Number>>()
{
  public int compare(Map.Entry<String, Number> o1,
                     Map.Entry<String, Number> o2)
  {
    return o1.getKey().compareTo(o2.getKey());
  }
};


/**
 * <p>Default maximum font size for rendered tags (20).</p>
 */
private static final int _DEFAULT_MAXIMUM_FONT_SIZE = 20;


/**
 * <p>Minimum font size for rendered tags (2).</p>
 */
private static final int _DEFAULT_MINIMUM_FONT_SIZE = 8;

/**
 * <p>The client event type for
 * <code>oracle.adfdemo.acme.js.event.AcmeTagSelectEvent.js</code>.
 * This constant will correspond to <code>AcmeTagSelectEvent.SELECT_EVENT_
 * TYPE</code>.</p>
 */
private static final String _CLIENT_SELECT_EVENT_TYPE = "tagSelect";

/**
 * <p>The tag parameter name on the
 * <code>oracle.asfdemo.acme.js.event.AcmeTagSelectEvent.js</code>.
 * The tag reports one of the weighted tags the user has click on.  The
```

```
 * <code>AddMarshalledProperties</code> function is where this parameter is
 * prepared to be sent to the server where it is intercepted by the
 * <code>oracle.adfinternal.view.faces.context.RichPhaseListener</code> and
 * unmarshaled into a <code>oracle.adf.view.rich.render.ClientEvent</code>.
 * </p>
 *
 */
private static final String _CLIENT_SELECT_EVENT_TAG_PARAM = "tag";

/**
 * <p>The root skinning key <code>acme|tagPane</code>.</p>
 */
private static final String _ROOT_STYLE_KEY = "acme|tagPane";

/**
 * <p>The maximum font size skinning key
 * <code>acme|tagPane-tr-maximum-font-size</code>.</p>
 */
private static final String _MAXIMUM_FONT_SIZE_PROPERTY =
  _ROOT_STYLE_KEY + "-tr-maximum-font-size";

/**
 * <p>The minimum font size skinning key
 * <code>acme|tagPane-tr-minimum-font-size</code>.</p>
 */
private static final String _MINIMUM_FONT_SIZE_PROPERTY =
  _ROOT_STYLE_KEY + "-tr-minimum-font-size";

/**
 * <p>TagPane content skinning key <code>acme|tagPane::content</code>.</p>
 */
private static final String _TAG_CONTENT_STYLE_KEY =
  "acme|tagPane::content";

/**
 * <p>Tag link skinning key <code>acme|tagPane::tag</code>.</p>
 */
private static final String _TAG_STYLE_KEY = "acme|tagPane::tag";

/**
 * <p>The <code>tags</code> property key from the <code>TagPane.TYPE</code>.</p>
 */
private static PropertyKey _TAGS_KEY = null;

/**
 * <p>The <code>orderBy</code> property key from the
 * <code>TagPane.TYPE</code>.</p>
 */
private static PropertyKey _ORDER_BY_KEY = null;

/**
 * <p>The Corresponding JavaScript component,
 * <code>oracle.adfdemo.acme.js.component.AcmeTagPane.js</code>.</p>
 */
private static final String _CLIENT_COMPONENT = "AcmeTagPane";

/**
 * <p>Pattern used to build the inline style that specifies the font
 * size for a tag.  Pixels units are assumed.</p>
 */
```

```
  private static final MessageFormat _FONT_INLINESTYLE_FORMAT =
    new MessageFormat("font-size:{0}px;");

}
```

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:

- Section G.1, "Introduction to Troubleshooting ADF Faces"
- Section G.2, "Getting Started with Troubleshooting the View Layer of an ADF Application"
- Section G.3, "Resolving Common Problems"
- Section G.4, "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.

G.1 Introduction to Troubleshooting ADF Faces

This section provides guidelines and a process for using the information in this chapter. 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	Section G.2	Get started troubleshooting the view layer of an ADF application. The procedures in this section quickly address a wide variety of problems.
2	Section G.3	Perform problem-specific troubleshooting procedures for the view layer of an ADF application. This section describes: ■ Possible causes of the problems ■ Solution procedures corresponding to each of the possible causes
3	Section G.4	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 and Community Forums and Discussion pages.
4	Section G.4	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`.

G.2 Getting Started with Troubleshooting the View Layer of an ADF Application

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 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, where as 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 look up error messages in *Error Messages*. You can also search the technical forums on Oracle Technology Network for discussions related to an error message. Each of the component layers for Oracle ADF has is 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.jspa?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.	Enable debug output by setting the following in the `trinidad-config.xml` file:
	`<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>`
	Improves the readability of HTML markup in the web browser:
	▪ 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.	Disable content compression by setting the following in the `web.xml` file:
	`<context-param>`
	`   <param-name>`
	`     org.apache.myfaces.trinidad.DISABLE_CONTENT_COMPRESSION`
	`   </param-name>`
	`   <param-value>true</param-value>`
	` </context-param>`
	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.

Disable JavaScript compression.	Disable JavaScript compression by setting the following in the `web.xml` file:
	`<context-param>`
	`   <param-name>`
	`     org.apache.myfaces.trinidad.DEBUG_JAVA_SCRIPT`
	`   </param-name>`
	`   <param-value>true</param-value>`
	` </context-param>`
	Allows normally obfuscated JavaScript to appear uncompressed as the source.

Enable client side asserts.	Enable client side asserts by setting the following in the `web.xml` file:
	`<context-param>`
	`   <param-name>`
	`     oracle.adf.view.rich.ASSERT_ENABLED`
	`   </param-name>`
	`   <param-value>true</param-value>`
	` </context-param>`
	Allows warnings of unexpected conditions to be output to the browser console.

Table G–2 (Cont.) Configuration Options for Optimizing ADF Faces Troubleshooting

Configuration Recommendation	Description
Enable clientside logging.	Enable clientside logging by setting the following in the web.xml file: <context-param> <param-name> **oracle.adf.view.rich.ASSERT_ENABLED** </param-name> <param-value>true</param-value> </context-param> 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.
Enable more detailed server side logging.	Enable more detailed server side logging shut down the application server, enter the following setting in the logging.xml file, and restart the server: <logger name="oracle.adf.faces” level=”CONFIG"/> or Use the WLST command: setLogLevel(logger="oracle.adf” level=”CONFIG”, addLogger=1) or In Oracle Enterprise Manager Fusion Middleware Control, use the Configuration page to set oracle.adf, oracle.adfinternal, and oracle.jbo to level CONFIG. 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.
Disable HTTP cache headers.	Disable HTTP cache headers by setting the following in the web.xml file: <context-param> <param-name> **org.apache.myfaces.trinidad.resource.DEBUG** </param-name> <param-value>true</param-value> </context-param> 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.

G.3 Resolving Common Problems

This section describes common problems and solutions.

G.3.1 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, as described in the "Applying an ADF Skin to Your Web Application" section in *Creating ADF Skins with Oracle ADF Skin Editor*.

G.3.2 Application is Missing Expected Images

The skin application must be packaged as a 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 JAR file, as described in the "Packaging an ADF Skin into an ADF Library JAR" section in *Creating ADF Skins with Oracle ADF Skin Editor*.

G.3.3 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 that 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 "Testing Changes in Your ADF Skin" section in *Creating ADF Skins with Oracle ADF Skin Editor*.

G.3.4 Data Visualization Components Fail to Display as Expected

Various ADF 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.

G.3.5 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 Java server system property `org.apache.myfaces.trinidad.CHECK_STATE_SERIALIZATION`. The following are Java 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
   ```

G.3.6 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 the 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 Tool's 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.

G.3.7 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. Recommend 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.

G.3.8 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 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.

G.3.9 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.

G.3.10 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 Section G.3.8, "Browser Displays an ADF_Faces-60098 Error."
2. Look for HTTP 404 or 505 errors, as described in Section G.3.9, "Browser Displays an HTTP 404 or 500 Error."

G.4 Using My Oracle Support for Additional Troubleshooting Information

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`.