Oracle® Fusion Middleware

Web User Interface Developer's Guide for Oracle Application Development Framework (Oracle Fusion Applications Edition)

11*g* Release 5 (11.1.1.6.3)

E28163-03

August 2012

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

ORACLE

Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework (Oracle Fusion Applications Edition), 11*g* Release 5 (11.1.1.6.3)

E28163-03

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

Contributing Author: Poh Lee Tan and Odile Sullivan-Tarazi

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

Contents

5 Handling Events

6 Validating and Converting Input

9 Using Input Components and Defining Forms

10 Using Tables and Trees

11 Using List-of-Values Components

12 Using Query Components

13 Using Popup Dialogs, Menus, and Windows

14 Using Menus, Toolbars, and Toolboxes

15 Creating a Calendar Application

16 Using Output Components

17 Displaying Tips, Messages, and Help

18 Working with Navigation Components

19 Creating and Reusing Fragments, Page Templates, and Components

20 Customizing the Appearance Using Styles and Skins

21 Internationalizing and Localizing Pages

22 Developing Accessible ADF Faces Pages

25 Using ADF Gauge Components

26 Using ADF Pivot Table Components

27 Using ADF Geographic Map Components

28 Using ADF Gantt Chart Components

29 Using ADF Hierarchy Viewer Components

Part V Advanced Topics

30 Creating Custom ADF Faces Components

31 Allowing User Customization on JSF Pages

32 Adding Drag and Drop Functionality

33 Using Different Output Modes

Part VI Appendixes

A ADF Faces Configuration

D Creating Web Applications for Touch Devices Using ADF Faces

E Quick Start Layout Themes

F Troubleshooting ADF Faces

Preface

Welcome to *Web User Interface Developer's Guide for Oracle Application Development Framework (Oracle Fusion Applications Edition)*!

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:

- *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework (Oracle Fusion Applications Edition)*
- *Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Mobile Browser Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*
- *Oracle JDeveloper 11g Online Help*

- *Oracle JDeveloper 11g Release Notes*, included with your JDeveloper 11*g* installation, and on Oracle Technology Network
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces Client JavaScript*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Data Visualization Components*
- *Oracle Fusion Middleware Tag Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces*
- *Oracle Fusion Middleware Tag Reference for Oracle ADF Faces Skin Selectors*
- *Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Skin Selectors*

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 for Release 11.1.1.6.3

For Release 11.1.1.6.3, this guide has been updated in several ways. The following table lists the sections that have been added or changed.

For changes made to Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) for this release, see the New Features page on the Oracle Technology Network at `http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html`.

Sections	Changes Made
Chapter 9 Using Input Components and Defining Forms	
Section 9.10, "Using Code Editor"	Section added to document the `codeEditor` component.
Chapter 12 Using Query Components	
Section 12.4.1, "How to Add the Query Component"	Added content about using the `queryAutomatically` property to control whether a query runs automatically.
Appendix A ADF Faces Configuration	
Section A.2.3.6, "Enabling the Application for Real User Experience Insight"	Section added
Section A.2.3.11, "Test Automation"	Section revised to document coding errors that will produce assertion failed errors only after test automation is enabled.
Appendix F Troubleshooting ADF Faces	
Appendix F, "Troubleshooting ADF Faces"	Appendix added to document best practices for troubleshooting ADF Faces applications.

Part I

Getting Started with ADF Faces

Part I contains the following chapters:

1

Introduction to ADF Faces Rich Client

This chapter introduces ADF Faces rich client, providing a history, an overview of the framework functionality, and an overview of each of the different component types found in the library. It also introduces the ADF Faces demonstration application that can be used in conjunction with this guide.

This chapter includes the following sections:

1.1 Introduction to ADF Faces Rich Client

ADF Faces rich client (known also as ADF Faces) is a set of JavaServer Faces (JSF) components that include built-in Asynchronous JavaScript and XML (AJAX) functionality. While AJAX brings rich client-like functionality to browser-based applications, using JSF provides server-side control, which reduces the amount of JavaScript code that application developers need to write in order to implement AJAX-based applications.

In addition to providing a rich set of JSF components, the ADF Faces rich client framework (RCF) provides a client-side programming model familiar to developers accustomed to the JSF development model. Most of the RCF differs little from any standard JSF application: the server programming model is still JavaServer Faces, and the framework still uses the JavaServer Faces lifecycle, server-side component tree, and the expression language (EL). However, the RCF also provides a client-side programming model and lifecycle that execute independently of the server. Developers can find and manipulate components from JavaScript, for example `get` and `set` properties, receive and queue events, and so forth, entirely from JavaScript. The RCF makes sure that changes to component state are automatically synchronized back to the server to ensure consistency of state, and that events are delivered, when necessary, to the server for further processing.

Before providing more detailed information regarding ADF Faces, it may help to have a brief history of the ADF Faces library and Rich Internet Applications (RIAs) and AJAX in general.

1.1.1 History of ADF Faces

In the 1990s, software vendors began to see the need for Internet applications to appear and behave more like desktop applications, and so they developed RIA

frameworks on which to build these applications. However, these frameworks required that users install proprietary plug-ins in order to utilize the functionality. As web standards developed, and Java web applications became more prevalent, the development community at large started to recognize the need for a standard view-layer framework. The Java Community Process (JCP) developed JSF as a user interface standard for Java web applications. From the formative years of JSR-127 in 2001, through the first release in 2004, and up to the current release, the JCP has brought together resources from the community, including Oracle, to define the JSF specification and produce a reference implementation of the specification. JSF is now part of the Java EE standard.

With JSF being a standard for building enterprise Java view components, vendors could now develop their own components that could run on any compliant application server. These components could now be more sophisticated, allowing developers to create browser-based RIAs that behaved more like thick-client applications. To meet this need, Oracle developed a set of components called ADF Faces that could be used on any runtime implementation of JSF. ADF Faces provided a set of over 100 components with built-in functionality, such as data tables, hierarchical tables, and color and date pickers, that exceeded the functionality of the standard JSF components. To underline its commitment to the technology and the open source community, Oracle has since donated that version of the ADF Faces component library to the Apache Software Foundation, and it is now known as Apache MyFaces Trinidad. This component library is currently available through the Apache Software Foundation.

ADF Faces not only provided a standard set of complex components, pages were now able to be partially refreshed using partial page rendering with AJAX. AJAX is a combination of asynchronous JavaScript, dynamic HTML (DHTML), XML, and the `XmlHttpRequest` communication channel, which allows requests to be made to the server without fully rerendering the page. However, pages built solely using AJAX require a large amount of JavaScript to be written by the developer.

The latest version of ADF Faces takes full advantage of AJAX, and it also provides a fully-functioning framework, allowing developers to implement AJAX-based RIAs relatively easily with a minimal amount of hand-coded JavaScript. Using ADF Faces, 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. ADF Faces insulates the developer from the need to deal with the intricacies of JavaScript and the DHTML differences across browsers.

1.1.2 ADF Faces as Rich Client Components

ADF Faces rich client framework offers complete RIA functionality, including drag and drop, lightweight dialogs, a navigation and menu framework, and a complete JavaScript API. The library provides over 100 RIA components, including hierarchical data tables, tree menus, in-page dialogs, accordion panels, dividers, and sortable tables. ADF Faces also includes data visualization components, which are Flash- and SVG-enabled components capable of rendering dynamic charts, graphs, gauges, and other graphics that provide a real-time view of underlying data. Each component also offers customizing and skinning, along with support for internationalization and accessibility.

To achieve these capabilities, ADF Faces components use a rich JSF render kit. This kit renders both HTML content as well as the corresponding client-side components. This

built-in support enables you to build RIAs without needing extensive knowledge of the individual technologies.

ADF Faces can also be used in an application that uses the Facelets library. Facelets is a JSF-centric declarative XML view definition technology that provides an alternative to using the JSP engine technology for the view. For more details about the architecture of ADF Faces, see Section 1.2, "Architecture of ADF Faces Components."

> **Tip:** You can use ADF Faces in conjunction with ADF Model data binding, allowing you to declaratively bind ADF Faces components to the business layer. Using ADF Model data binding, most developer tasks that would otherwise require writing code are declarative. However, this guide covers only using ADF Faces components in a standard JSF application. For more information about using ADF Faces with the ADF Model, see the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

In addition to an extensive library of RIA components, Oracle also offers Oracle JDeveloper, a full-featured development environment with built-in declarative support for ADF Faces components, allowing you to quickly and easily build the view layer of your web application. JDeveloper contains a visual layout editor that displays JSF pages in a WYSIWYG environment. The Component Palette in JDeveloper holds visual representations of each of the ADF Faces components, which allows you to drag and drop a component onto a page in the visual editor, instead of having to manually add tag syntax to a page. You can use JDeveloper throughout the complete development lifecycle, as it has integrated features for modeling, coding, debugging, testing, tuning, and deploying. For more information about using JDeveloper, see Chapter 2, "Getting Started with ADF Faces."

1.2 Architecture of ADF Faces Components

Unlike frameworks where most of the application logic resides on the client, with ADF Faces application logic resides mostly on the server, executing in the JSF lifecycle. The Java data model also remains on the server: the ADF Faces framework performs initial rendering of its components on the server, generating HTML content that is consumed directly by browsers. Rendering HTML on the server means that there is less client-side rendering overhead, which is helpful for complex components.

> **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, it is recommended that you have a basic understanding of how JSF works before beginning to develop with ADF Faces. To learn more about JSF, visit the Java web site at `http://www.oracle.com/technetwork/java/index.html`.

1.2.1 Client-Side Architecture

JavaScript performance can suffer when too many objects are created. To improve performance, the RCF minimizes the number of component objects present on the client, and the number of attributes sent to the client. The framework also has the JavaScript files that make up the components housed in configurable partitions, allowing your application to load only the required JavaScript.

1.2.1.1 Client-Side 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 still exists on the server, the ADF Faces client-side component tree is sparsely populated. 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. Therefore, client-side components are created only for those components that are truly needed on the client, typically those that have been explicitly configured to have client representation.

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 Java `RichPopup` component may exist.

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. Each client component has a set of properties (key/value pairs) and a list of listeners for each supported event type. All RCF JavaScript classes are prefixed with `Adf` to avoid naming conflicts with other JavaScript libraries. For example, `RichCommandButton` has `AdfRichCommandButton`, `RichDocument` has `AdfRichDocument`, and so on.

While the Java `UIComponent` object represents the state of the component, and this object is what you interact with to register listeners and set properties, the `Renderer` handles producing HTML and receiving postbacks on behalf of the component. In the RCF client-side JavaScript layer, client-side components 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

1.2.1.2 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. For more information about configuring JavaScript partitioning, see Section A.9, "Using JavaScript Library Partitioning."

1.2.2 ADF Faces Architectural Features

The RCF enables many architectural features that can be used throughout your application. For example, because processing can be done on the client, small amounts of data can be exchanged with the server without requiring the whole page to be rendered. This is referred to as *partial page rendering* (PPR). Many ADF Faces components have PPR functionality implemented natively. For example, the ADF Faces table component comes with built-in AJAX-style functionality that 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 a roundtrip to the server, and with no coding needed. For more information, see Chapter 7, "Rerendering Partial Page Content."

The RCF also adds functionality to the standard JSF lifecycle. Examples include a client-side value lifecycle, a subform component that allows you to create independent submittable regions on a page without the drawbacks of using multiple forms on a single page, and an optimized lifecycle that can limit the parts of the page submitted for processing. For more information, see Chapter 4, "Using the JSF Lifecycle with ADF Faces."

The RCF uses the standard JSF event framework. However, events in the RCF have been abstracted from the standard JavaScript DOM event model. Though the events share some of the same abstractions found in the DOM event model, they also add functionality. Consequently, you need not listen for *click* events on buttons, for example. You can instead listen for `AdfActionEvent` events, which may or may not have been caused by key or mouse events. RCF events can be configured to either deliver or not deliver the event to the server. For more information, see Chapter 5, "Handling Events."

ADF Faces input components have built-in validation capabilities. You set one or more validators on a component by either setting the `required` attribute or by using the prebuilt ADF Faces validators. In addition, you can create your own custom validators to suit your business needs.

ADF Faces input components also have built-in conversion capabilities, which allow users to enter information as a string and the application can automatically convert the string to another data type, such as a date. Conversely, data stored as something other than a string can be converted to a string for display and updating. Many components, such as the `inputDate` component, automatically provide this capability. For more information, see Chapter 6, "Validating and Converting Input."

In addition to these architectural features, the RCF also supports the following:

- Fully featured client-side architecture: Many of the features you need to create AJAX-type functionality in your web application are found in the client side of the architecture. For more information, see Chapter 3, "Using ADF Faces Architecture."

- Reuse: 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 19, "Creating and Reusing Fragments, Page Templates, and Components."
- Skinning: You can globally change the appearance of components. For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins."
- Internationalization: You can change text and other display attributes based on the user's locale. For more information, see Chapter 21, "Internationalizing and Localizing Pages."
- Accessibility: You can implement accessibility support, including keyboard shortcuts and text descriptions. For more information, see Chapter 22, "Developing Accessible ADF Faces Pages."
- Custom component creation: You can create your own components that use the RCF. For more information, see Chapter 30, "Creating Custom ADF Faces Components."
- User customizations: You can create your pages so that they allow users to change certain display attributes for components at runtime. For more information, see Chapter 31, "Allowing User Customization on JSF Pages."
- Drag and drop: You can allow attribute values, collection values, or complete components to be dragged from one component to another. For more information, see Chapter 32, "Adding Drag and Drop Functionality."

1.3 ADF Faces Components

ADF Faces components generally fall into two categories. Layout components are those that are used to organize the contents of the page. Along with components that 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. Certain layout components support geometry management, that is, the process by which the size and location of components appear on a page. The RCF notifies these 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. 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 8, "Organizing Content on Web Pages."

The remaining components are considered to be in the common category, and are divided into the following subcategories:

- 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 9, "Using Input Components and Defining Forms."
- 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 10, "Using Tables and Trees."

- 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 11, "Using List-of-Values Components."
- Query components: Allow users to query data. ADF Faces provides two query components. 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. The `QuickQuery` component is a simplified version of the `Query` component that allows a search on a single item (criterion). For more information, see Chapter 12, "Using Query Components."
- Popup components: 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 13, "Using Popup Dialogs, Menus, and Windows."
- Explorer-type menus and toolbars: 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 14, "Using Menus, Toolbars, and Toolboxes."
- 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 15, "Creating a Calendar Application."
- Output components: Display text and graphics, and can also play video and music clips. ADF Faces also includes a `carousel` output component that can display graphics in a revolving carousel. For more information, see Chapter 16, "Using Output Components."
- Labels, tips, and messages: Display labels for other components, along with mouseover tips and error messages. Unlike standard JSF input components, ADF Faces components that support messages automatically display their own messages. You can also have components display informational content, for example contextual help. For more information, see Chapter 17, "Displaying Tips, Messages, and Help."
- 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 18, "Working with Navigation Components."
- Data visualization components: Allow users to view and analyze complex data in real time. ADF data visualization components include graphs, gauges, pivot tables, geographic maps, Gantt charts, and hierarchy viewers that display hierarchical data as a set of linked nodes, for example an organization chart. For more information, see Chapter 23, "Introduction to ADF Data Visualization Components."

1.4 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.

1.4.1 How to Download and Install 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 link for installing the ADF Faces Rich Client demo. 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.

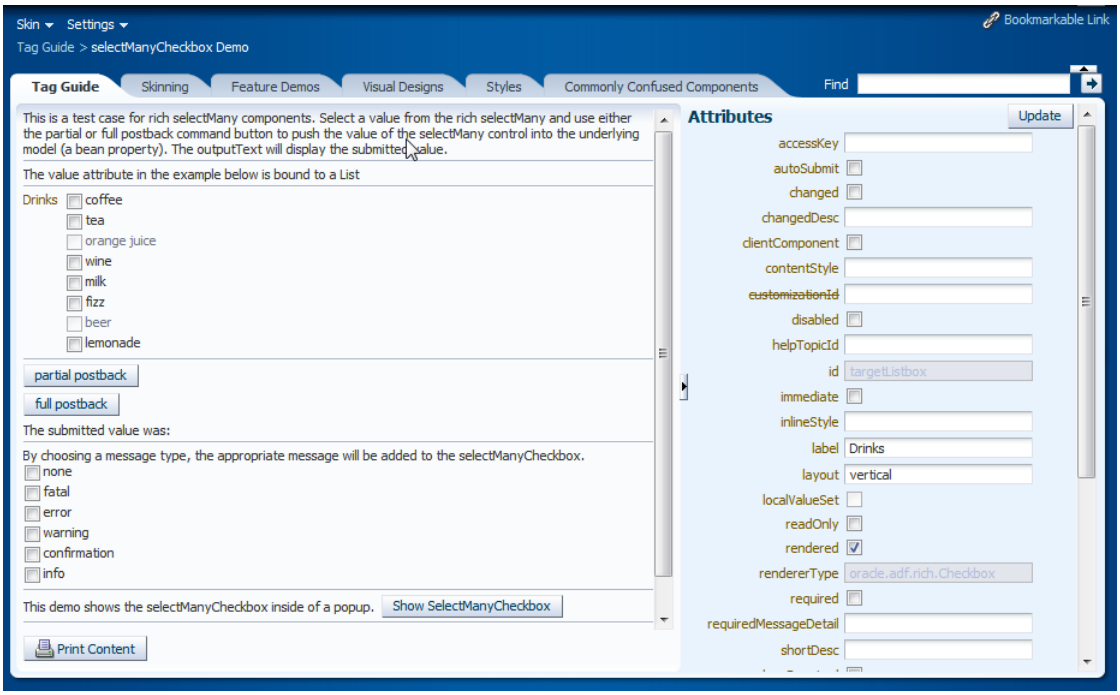
If you do not want to install the application, you can run the application directly from OTN by clicking the **ADF Faces Rich Client Components Hosted Demo** link.

1.4.2 Using the ADF Faces Demo Application

The demo application contains the following:

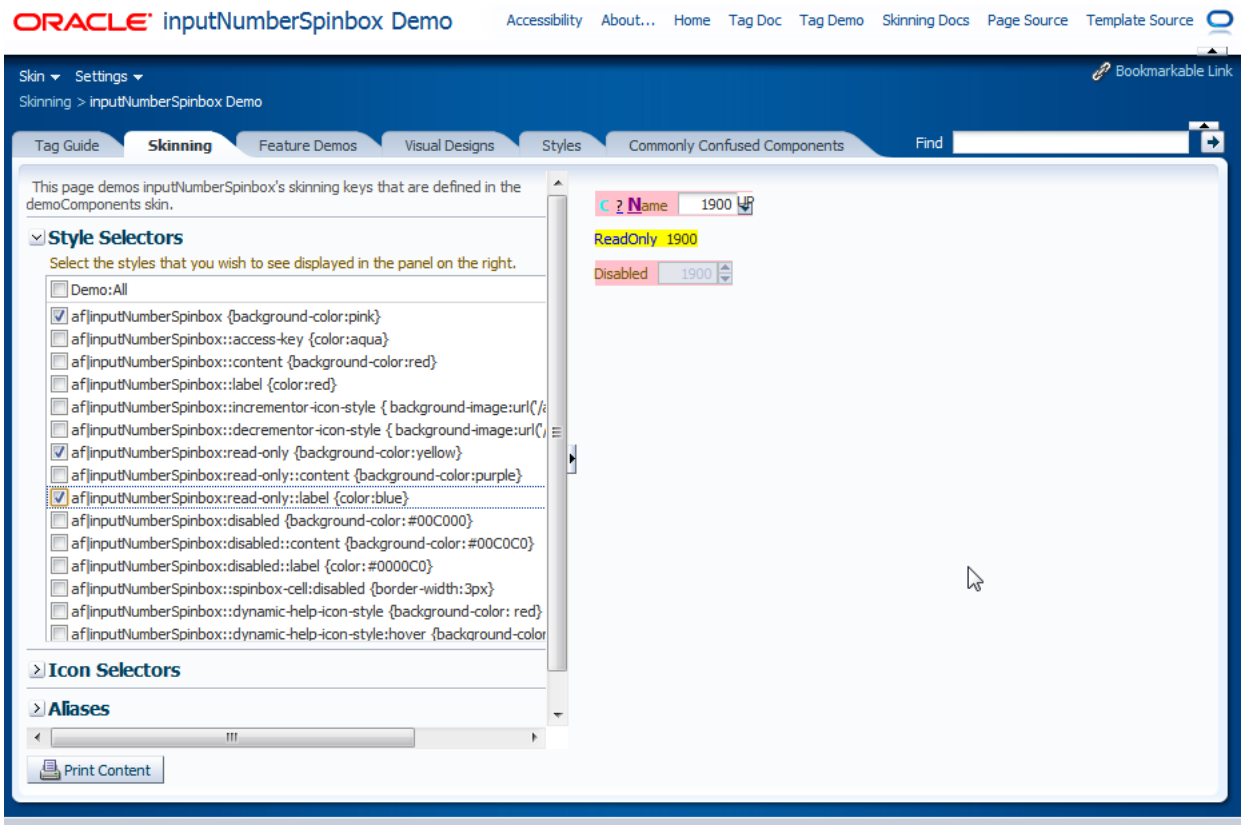
- Tag guide: Demonstrations of ADF Faces components, validators, converters, and miscellaneous tags, along with a property editor to see how changing attribute values affects the component. Figure 1–1 shows the demonstration of the `selectManyCheckbox` component. Each demo provides a link to the associated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

Figure 1–1 Tag Demonstration



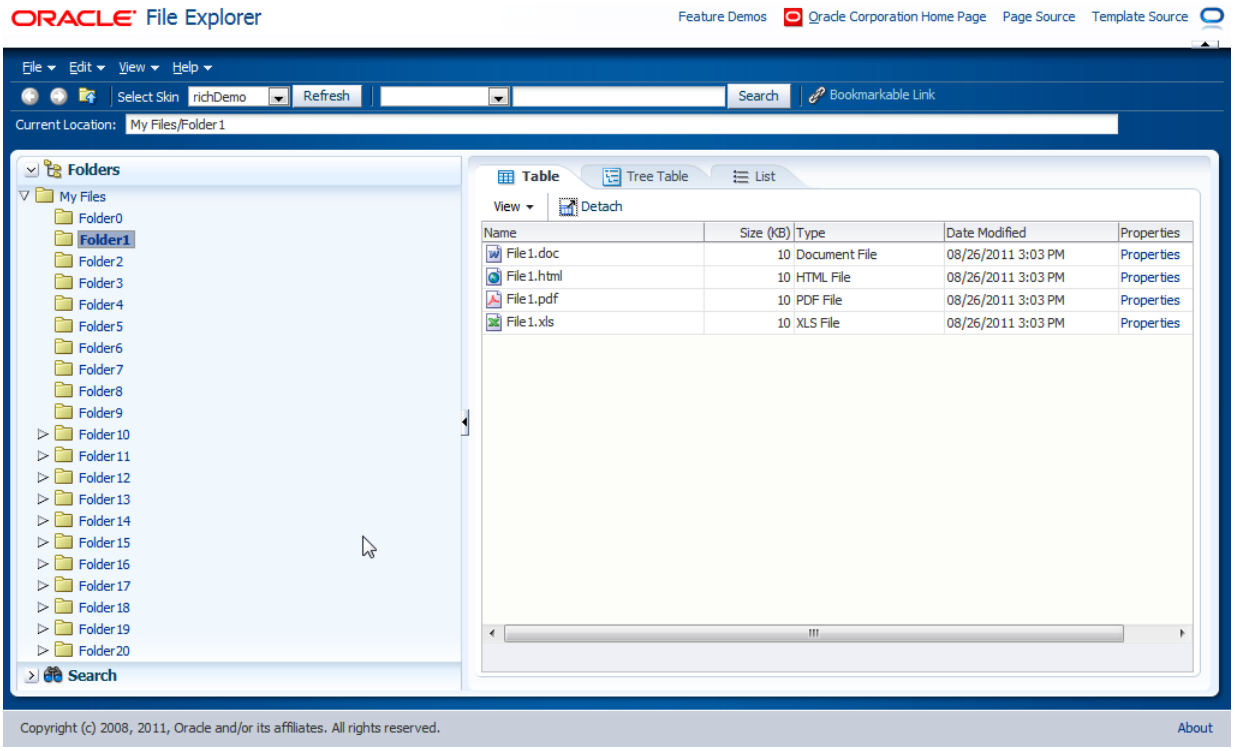
- Skinning: Demonstrations of skinning on the various components. You can see, for example, how changing style selectors affects how a component is displayed. Figure 1–2 shows how setting certain style selectors affects the `inputNumberSpinbox` component.

Figure 1–2 Skinning Demonstration



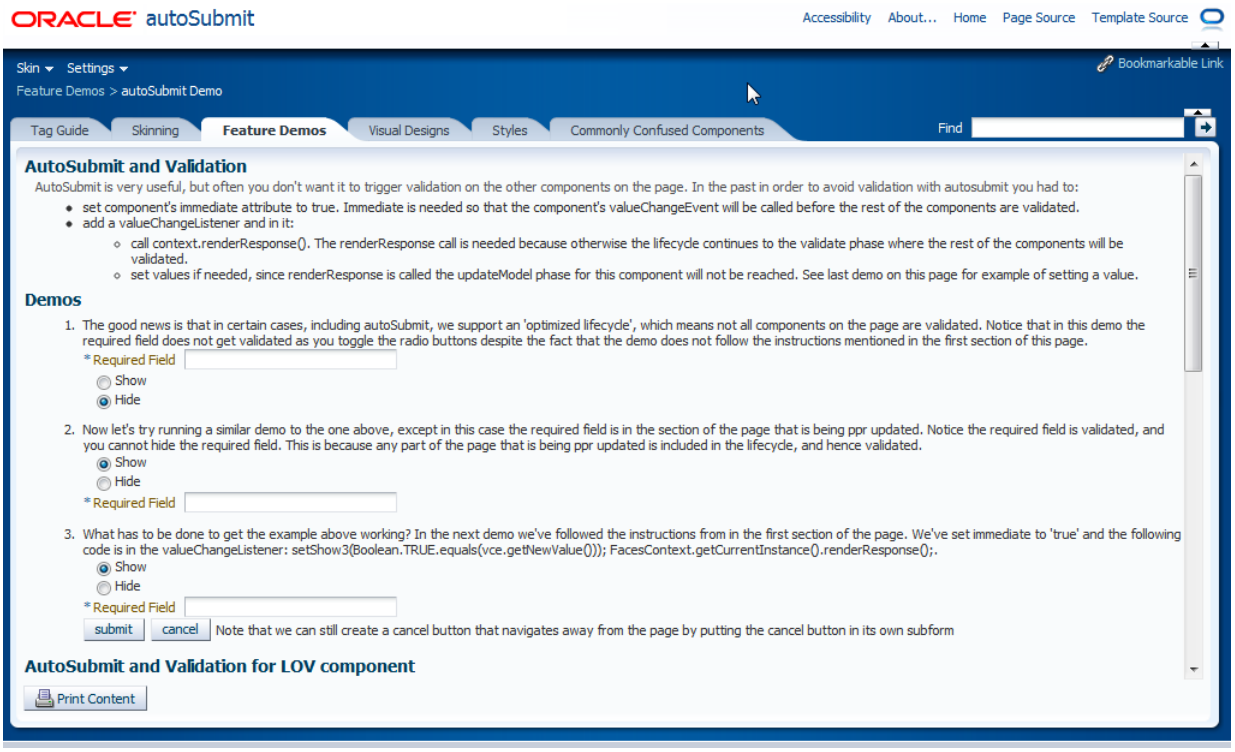
- Feature demos: Various pages that demonstrate different ways you can use ADF components. For example, the File Explorer is an application with a live data model that displays a directory structure and allows you to create, save, and move directories and files. This application is meant to showcase the components and features of ADF Faces in a working application, as shown in Figure 1–3. For more information about the File Explorer application, see Section 1.4.3, "Overview of the File Explorer Application."

Figure 1–3 File Explorer Application



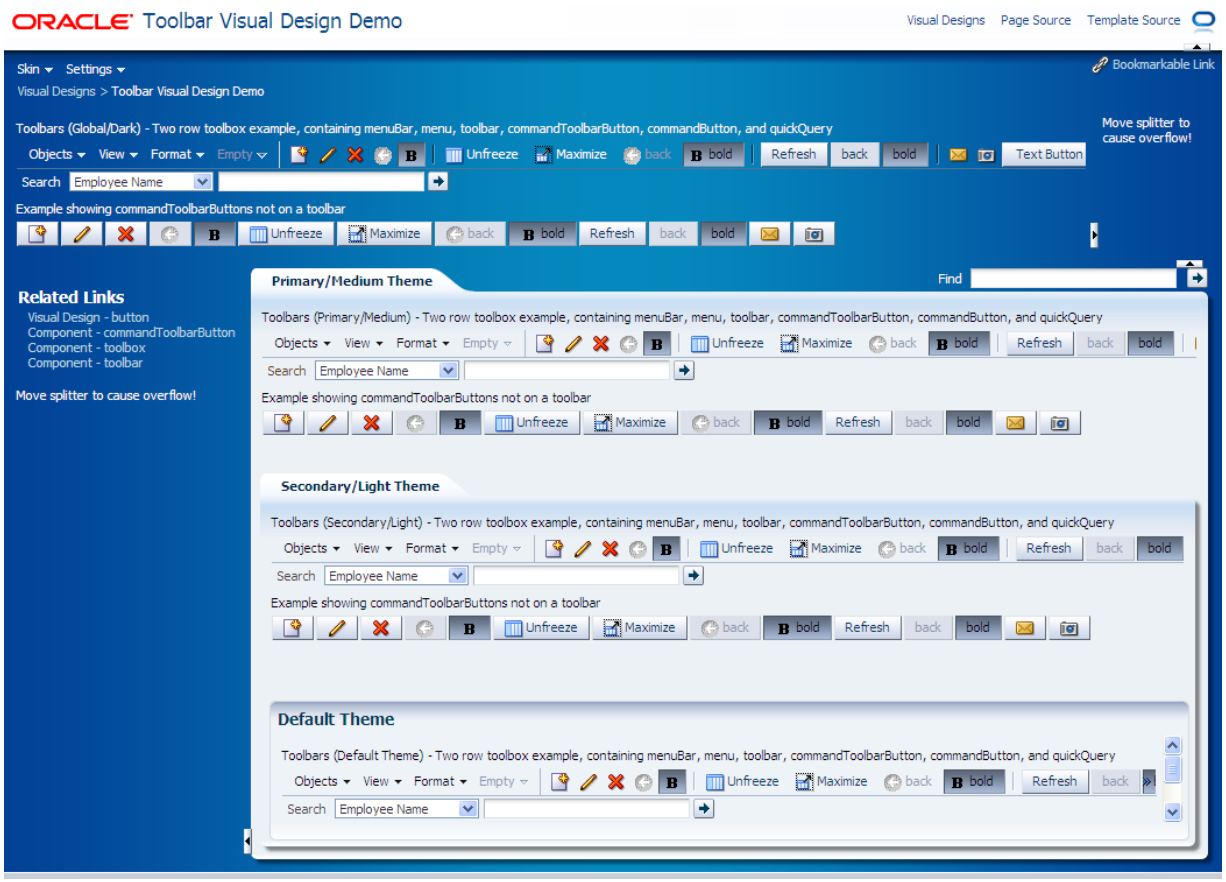
Other pages demonstrate the main architectural features of ADF Faces, such as layout components, AJAX postback functionality, and drag and drop. Figure 1–4 shows the demonstration on using the `AutoSubmit` attribute and validation.

Figure 1–4 Framework Demonstration



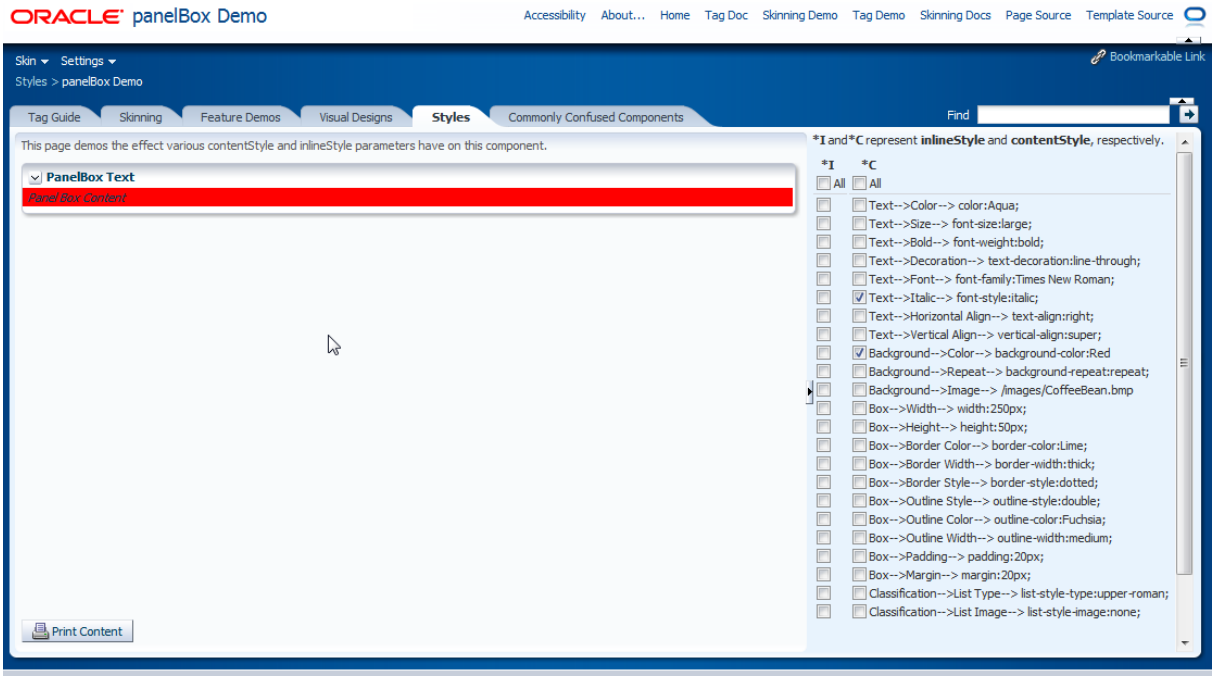
- Visual designs: Demonstrations of how you can use types of components in different ways to achieve different UI designs. Figure 1–5 shows how you can achieve different looks for a toolbar.

Figure 1–5 Toolbar Design Demonstration



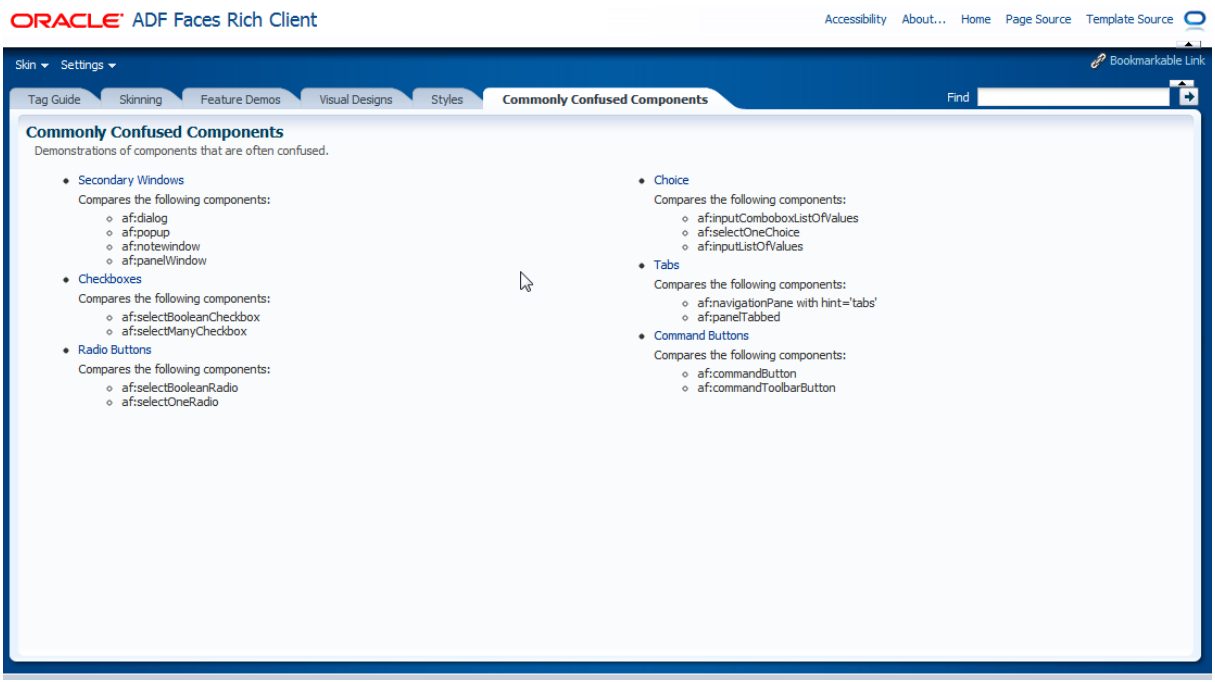
- Styles: Demonstration of how setting inline styles and content styles affects components. Figure 1–6 shows different styles applied to the `panelBox` component.

Figure 1–6 Styles Demonstration



- Commonly confused components: A comparison of components that provide similar functionality. Figure 1–7 shows the differences between the various components that display lists.

Figure 1–7 Commonly Confused Components



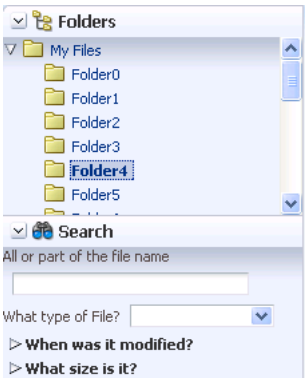
1.4.3 Overview of the File Explorer Application

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

The File Explorer application uses the `fileExplorerTemplate` page template. This template contains a number of layout components that provide the basic look and feel for the application. For more information about layout components, see Chapter 8, "Organizing Content on Web Pages." For more information about using templates, see Chapter 19, "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 1–8.

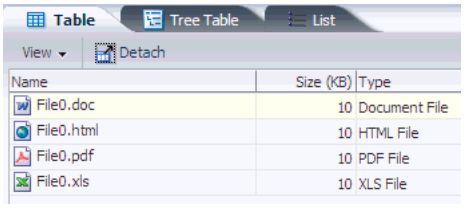
Figure 1–8 Directory Structure Panel and Search Panel



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 command button, and a `table` component. For more information about using `panelAccordion` components, see Section 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed Panels." For more information about using input components, see Chapter 9, "Using Input Components and Defining Forms." For more information about using command buttons, see Chapter 18, "Working with Navigation Components." For more information about using tables and trees, see Chapter 10, "Using Tables and Trees."

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 1–9.

Figure 1–9 Directory Contents in Tabbed Panels

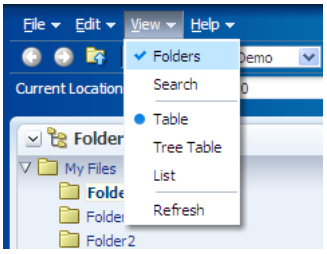


The table and tree table have built-in toolbars that allow you to manipulate how the contents are displayed. In the table an list, you can drag a file or subdirectory from one directory and drop it into another. In all tabs, you can right-click a file, and from the context menu, you can view the properties of the file in a popup window. For more

information about using tabbed panes, see Section 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed Panels." For more information about table and tree table toolbars, see Section 10.8, "Displaying Table Menus, Toolbars, and Status Bars." For more information about enabling drag and drop, see Chapter 32, "Adding Drag and Drop Functionality." For more information about using context menus and popup windows, see Chapter 13, "Using Popup Dialogs, Menus, and Windows."

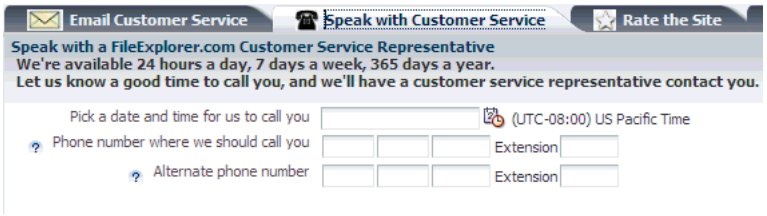
The top of the File Explorer application contains a menu and a toolbar, as shown in Figure 1–10.

Figure 1–10 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 1–11.

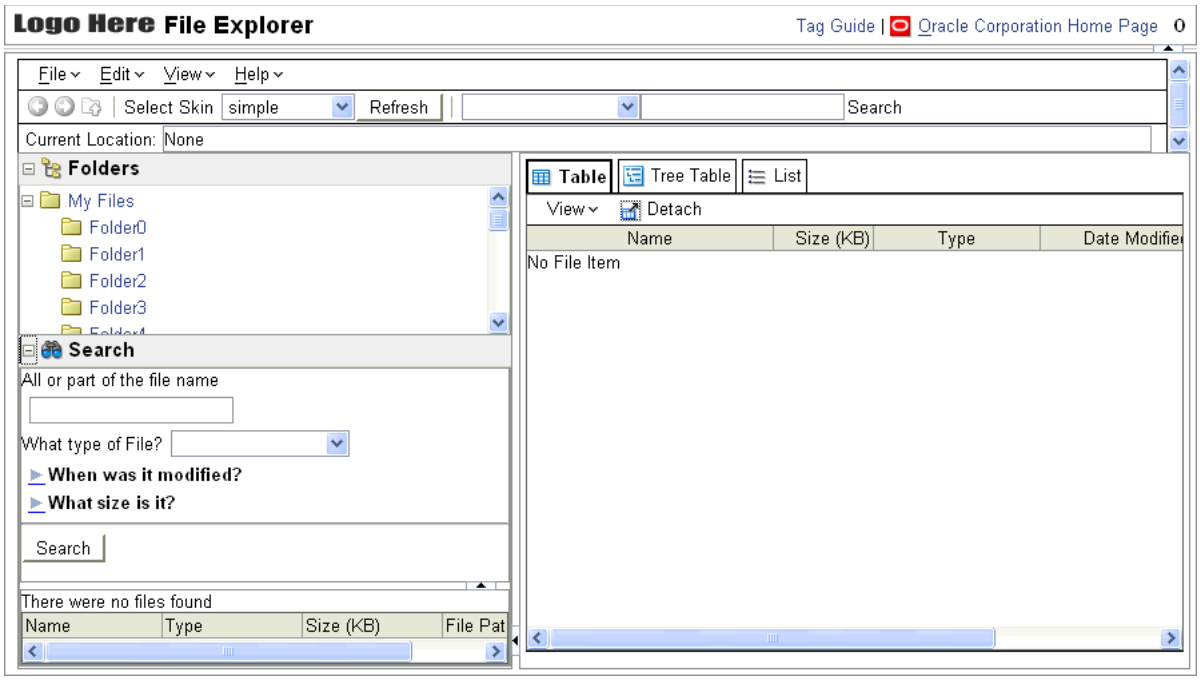
Figure 1–11 Help System



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 14.2, "Using Menus in a Menu Bar." For more information about creating help systems, see Section 17.5, "Displaying Help for Components." For more information about input components, see Chapter 9, "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 1–12 shows the File Explorer application using the simple skin.

Figure 1–12 File Explorer Application with the Simple Skin



For more information about toolbars, see Section 14.3, "Using Toolbars." For more information about using skins, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

1.4.4 Viewing the Source Code In JDeveloper

All the source files for the ADF Faces demo application are contained in one project (you give this project a name when you create it during installation). The project is divided into two directories: Application Sources and Web Content. Application Sources contains the `oracle.adfdemo.view package`, which in turn contains packages that hold managed beans that provide functionality throughout the application.

> **Tip:** The managed beans for the component demos are in the `component` package and the managed beans for the File Explorer application are in the `explorer` package.

The Web Content directory contains all the web resources used by the application, including JSPX files, JavaScript libraries, images, configuration files, and so on.

> **Tip:** The `components` subdirectory contains the resources for the component demos. The `docs` directory contains the tag and Javadoc documentation. The `fileExplorer` directory contains the resources for the File Explorer application.

2

Getting Started with ADF Faces

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

This chapter includes the following sections:

2.1 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 JavaServer Pages (JSPs) or Facelet pages
- Deploying the application. For more information about deployment, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*. If your application uses ADF Faces with the ADF Model layer, the ADF Controller, and ADF Business Components, see the "Deploying Fusion Web Applications" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Ongoing tasks throughout the development cycle will probably 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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.2 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. By creating an application using application templates provided by JDeveloper, you automatically get the organization of your workspace into projects, along with many of the configuration files and libraries required by the type of application you are creating.

2.2.1 How to Create an Application Workspace

You create an application workspace using the Create Application wizard.

To create an application:

1. In the JDeveloper main menu, choose **File > New**.

   The New Gallery opens, where you can select different application components to create.

2. In the New Gallery, expand the **General** node, select **Applications** and then **Java EE Web Application**, and click **OK**.

   This template provides the building blocks you need to create a web application that uses JSF for the view and Enterprise JavaBean (EJB) session beans and Java Persistence API (JPA) entities for business services. All the files and directories for the business layer of your application will be stored in a project that by default is named `Model`. All the files and directories for your view layer will be stored in a project that by default is named `ViewController`.

   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, the ADF Controller, and ADF Business Components, see the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*. For more information about using ADF Faces with the ADF Model layer and EJBs and JPA, see Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework.

3. In the Create Java EE Web 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 web project. On the **Project Technologies** tab, double-click **ADF Faces** to move that technology to the **Selected** pane. This automatically adds the necessary libraries and metadata files to your web project. Click **Next**.

5. In the Configure Java Settings page, optionally change the package name, Java source path, and output directory for your view layer. Click **Next**.

6. In the Name Your Project page, you can optionally change the name and location for your Java project. By default, the necessary libraries and metadata files for Java EE are already added to your data model project. Click **Next**.
7. In the Configure Java Settings page, optionally change the package name, Java source path, and output directory for your model layer. Click **Next**.
8. Configure the EJB settings as needed. For help on this page, click **Help** or press **F1**. Click **Finish**.

2.2.2 What Happens When You Create an Application Workspace

When you create an application workspace using the Java EE Web Application template, JDeveloper creates a project named `Model` that will contain all the source files related to the business services in your application. JDeveloper automatically adds the libraries needed for your EJB project. For example, if you kept the default EJB settings, JDeveloper adds the EJB 3.0 library.

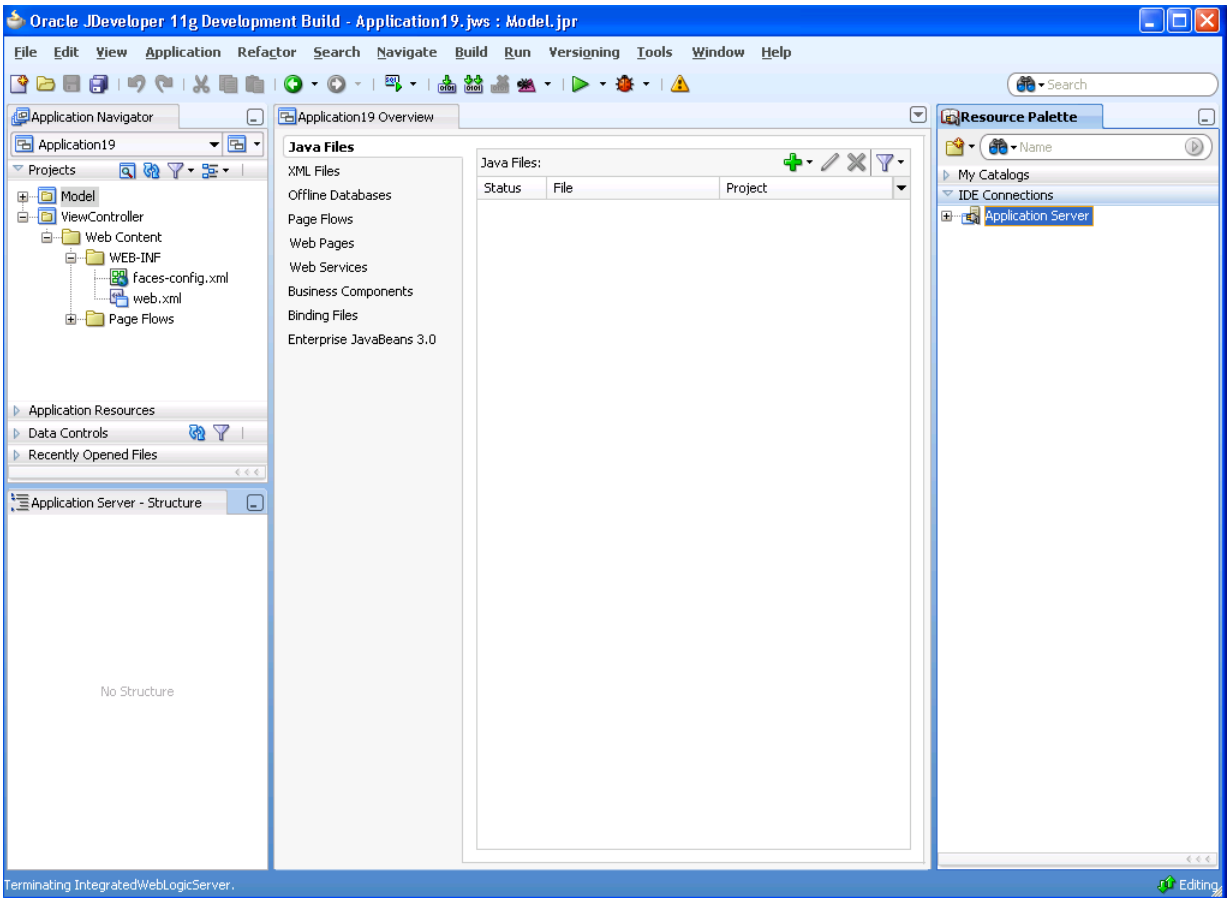
JDeveloper also creates a project named `ViewController` that will contain all the source files for your ADF Faces view layer. JDeveloper automatically creates the JSF and ADF configuration files needed for the application. Additionally, JDeveloper adds the following libraries to your view project:

- JSF 1.2
- JSTL 1.2
- JSP Runtime

The ADF Faces and other runtime libraries are added when you create a JSF page in your project.

Once the projects are created for you, you can rename them. Figure 2–1 shows the workspace for a new Java EE Web application.

Figure 2–1 New Workspace for an ADF 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 2–1 shows the `web.xml` file generated by JDeveloper when you create a new Java EE application.

Example 2–1 Generated web.xml File

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<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>
</web-app>
```

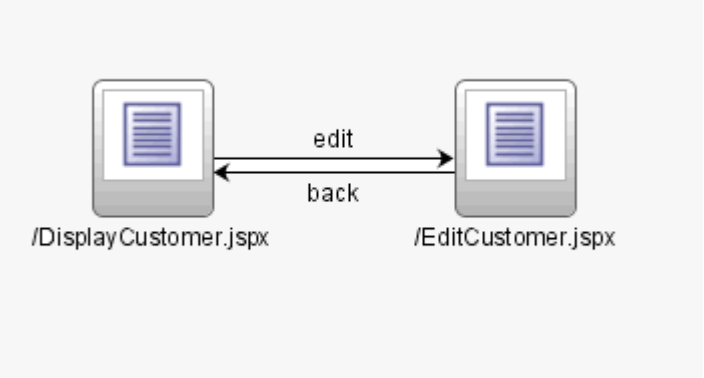
Configurations required for specific ADF Faces features are covered in the respective chapters of this guide. For example, any configuration needed in order to use the Change Persistence framework is covered in Chapter 31, "Allowing User Customization on JSF Pages." For comprehensive information about configuring an ADF Faces application, see Appendix A, "ADF Faces Configuration."

2.3 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 2–2 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 2–2 Navigation Diagram in JDeveloper



Note: If you plan on using ADF Model data binding and the ADF Controller, then instead of using standard JSF navigation rules, you use task flows. For more information, see the "Getting Started With ADF Task Flows" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Best Practice: The 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.

For more information on how navigation works in a JSF application, see the Java EE 5 tutorial at `http://www.oracle.com/technetwork/java/index.html`.

2.3.1 How to Define a Page Flow

You use the navigation diagrammer to declaratively create a page flow using JSP 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.

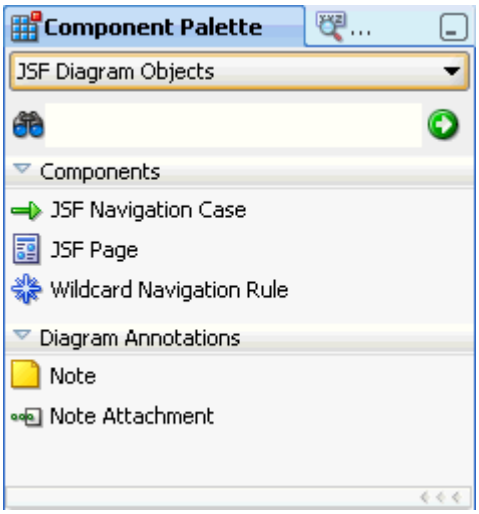
Note: The diagrammer supports only pages created as JSP and JSPX files. If you need to create navigation for XHTML pages, you must code the XML manually.

To create a page flow:

1. In the Application Navigator, double-click the `faces-config.xml` file for your application. By default, this is in the **Web Content**/**WEB-INF** node.
2. In the editor window, click the **Diagram** tab to open the navigation diagrammer.
3. If the Component Palette is not displayed, from the main menu choose **View > Component Palette**. By default, the Component Palette is displayed in the upper right-hand corner of JDeveloper.
4. In the Component Palette, use the dropdown menu to choose **JSF Diagram Objects**.

   The components are contained in two accordion panels: **Components** and **Diagram Annotations**. Figure 2–3 shows the Component Palette displaying JSF navigation components.

Figure 2–3 Component Palette 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. Press **F1** for details on using the overview editor to create navigation.

   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 18, "Working with Navigation Components."

2.3.2 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 2–2 shows the XML generated for the navigation rules displayed in Figure 2–2.

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

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

2.4 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 JSP files. When you create a JSP for an ADF Faces application, you can choose to create an XML-based JSP document (which uses the extension `*.jspx`) rather than a `*.jsp` file.

> **Best Practice:** 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.

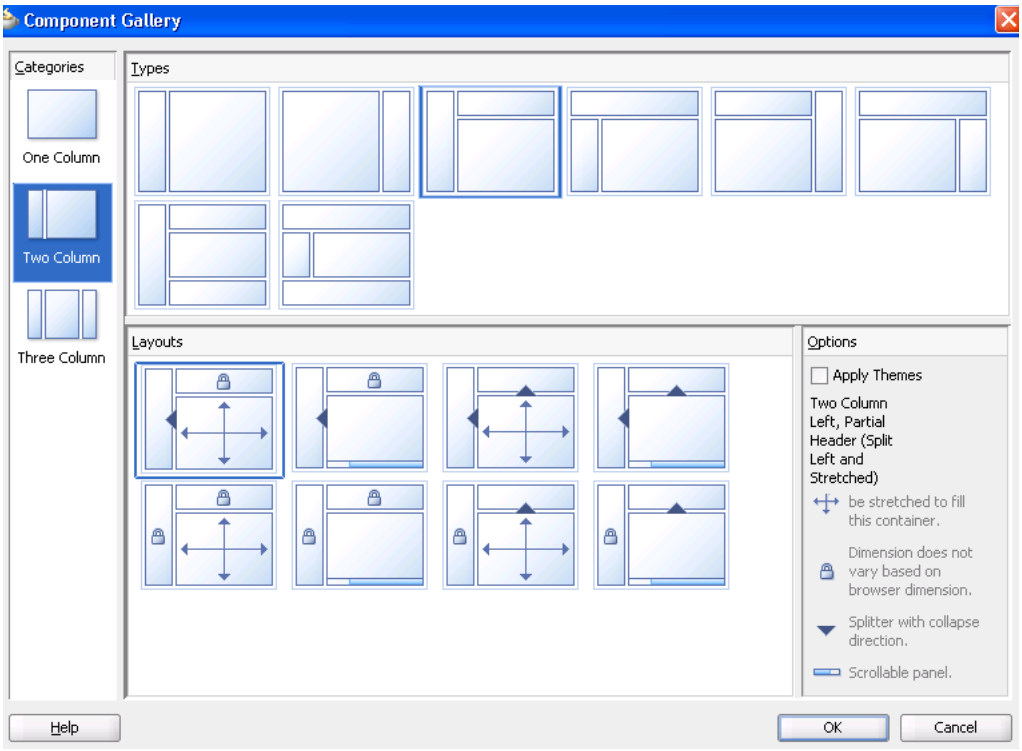
If you want to use Facelets instead of JSP in your application, you can instead create XHTML files. Facelets is a JSF-centric declarative XML view definition technology that provides an alternative to using the JSP engine.

> **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 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 2–4 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 8, "Organizing Content on Web Pages."

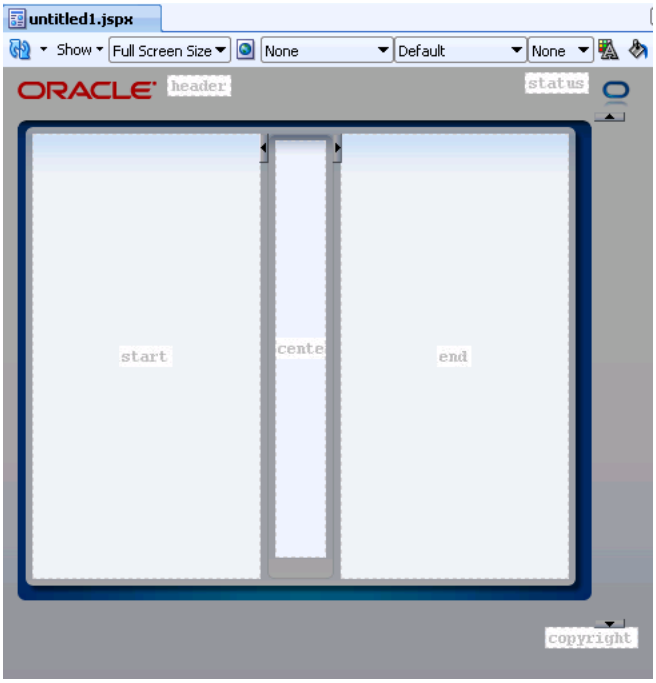
Figure 2–4 Quick Layouts



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 Section 20.3.4, "How to Apply Themes to Components."

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 (either by selecting a quick layout design, as shown in Figure 2–4, or by building it manually) 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 2–5.

Figure 2–5 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 19.3, "Using Page Templates."

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 JSP pages, see Section 2.4.3, "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).

You can also choose to have your page available for display in mobile devices. Once your page files are created, you can add UI components and work with the page source.

2.4.1 How to Create JSF JSP Pages

You create JSF JSP pages using the Create JSF Page dialog.

To create a JSF JSP page:

1. In the Application Navigator, right-click the directory where you would like the page to be saved, and choose **New**. In the New Gallery, expand the **Web Tier** node, select **JSF** and then **JSF Page,** and click **OK**.

   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 Page Implementation option, which can be used to automatically create a backing bean and associated bindings, see Section 2.4.3, "What You May Need to Know About Automatic Component Binding."

2.4.2 What Happens When You Create a JSF JSP 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 `.jspx` document. Example 2–3 shows a `.jspx` page when it is first created by JDeveloper.

Example 2–3 Declarative 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:h="http://java.sun.com/jsf/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document 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 2–4 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 2–4 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:h="http://java.sun.com/jsf/html"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document id="d1">
      <af:form id="f1">
        <af:panelStretchLayout startWidth="100px" id="psl1">
          <f:facet name="start"/>
          <f:facet name="center">
            <!-- id="af_twocol_left_sidebar_stretched"   -->
            <af:decorativeBox theme="dark" id="db2">
              <f:facet name="center">
                <af:decorativeBox theme="medium" id="db1">
                  <f:facet name="center"/>
                </af:decorativeBox>
              </f:facet>
            </af:decorativeBox>
          </f:facet>
        </af:panelStretchLayout>
      </af:form>
```

```
    </af:document>
  </f:view>
</jsp:root>
```

If you chose to automatically create a backing bean using the Page Implementation section of the dialog, JDeveloper also creates and registers a backing bean for the page, and binds any existing components to the bean. Example 2–5 shows the code created for a backing bean for a page.

Example 2–5 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
- Oracle JEWT
- ADF DVT Faces Databinding Runtime

JDeveloper also adds entries to the `web.xml` file, as shown in Example 2–6.

Example 2–6 Code in the web.xml File After a JSF JSP Page is Created

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<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">
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION</param-name>
  <param-value>false</param-value>
</context-param>
<context-param>
  <param-name>oracle.adf.view.rich.versionString.HIDDEN</param-name>
  <param-value>false</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>
</filter-mapping>
<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.adfinternal.view.faces.bi.renderkit.graph.GraphServlet
  </servlet-class>
</servlet>
<servlet>
  <servlet-name>BIGAUGESERVLET</servlet-name>
  <servlet-class>
   oracle.adfinternal.view.faces.bi.renderkit.gauge.GaugeServlet
  </servlet-class>
</servlet>
<servlet>
  <servlet-name>MapProxyServlet</servlet-name>
  <servlet-class>
    oracle.adfinternal.view.faces.bi.renderkit.geoMap.servlet.MapProxyServlet
  </servlet-class>
</servlet>
<servlet>
  <servlet-name>GatewayServlet</servlet-name>
  <servlet-class>
    oracle.adfinternal.view.faces.bi.renderkit.graph.FlashBridgeServlet
  </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>
  <servlet-mapping>
    <servlet-name>GatewayServlet</servlet-name>
    <url-pattern>/flashbridge/*</url-pattern>
  </servlet-mapping>
</web-app>
```

In the `faces-config.xml` file, when you create a JSF JSP 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 2–7.

Example 2–7 Generated 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>
    <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 2–8.

Example 2–8 Generated trinidad-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>fusion</skin-family>
</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.

2.4.3 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 2.6, "Creating and Using Managed Beans"). If when you create your JSF JSP 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:

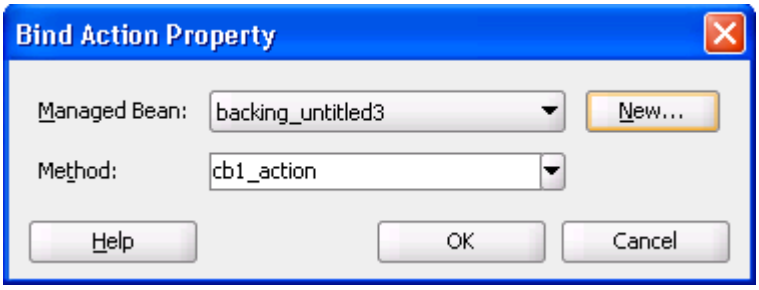
- Creates a JavaBean using the same name as the JSP or JSPX file, and places it in the `view.backing` package (if you elect to have JDeveloper create a backing bean).
- Creates a managed bean entry in the `faces-config.xml` file for the backing bean. By default, the managed bean name is `backing_<page_name>` and the bean uses the `request` scope (for more information about scopes, see Section 4.6, "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 JSP. 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 JSP 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 JSP 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 command 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 2–6.

Figure 2–6 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 JSP, you must select an existing managed bean or create a new backing bean to create the binding.

For example, suppose you created a JSP with the file name `myfile.jspx`. 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 2–9 shows the code on a JSP that uses automatic component binding, and contains `form`, `inputText`, and `commandButton` components.

Example 2–9 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.inputText1}"
                    id="inputText1"/>
      <af:commandButton text="commandButton 1"
                        binding="#{backing_MyFile.cb1}"
                        id="cb1"/>
    </af:form>
  </af:document>
</f:view>
```

Example 2–10 shows the corresponding code on the backing bean.

Example 2–10 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.RichCommandButton;

public class MyFile {
    private RichForm f1;
    private RichDocument d1;
    private RichInputText inputText1;
    private RichCommandButton cb1;

    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 inputText1) {
        this.inputText1 = inputText1;
    }

    public RichInputText getInputText1() {
        return inputText1;
    }

    public void setCb1(RichCommandButton cb1) {
        this.commandButton1 = commandButton1;
    }

    public RichCommandButton getCb1() {
        return cb1;
    }

    public String cb1_action() {
        // Add event code here...
        return null;
    }
}
```

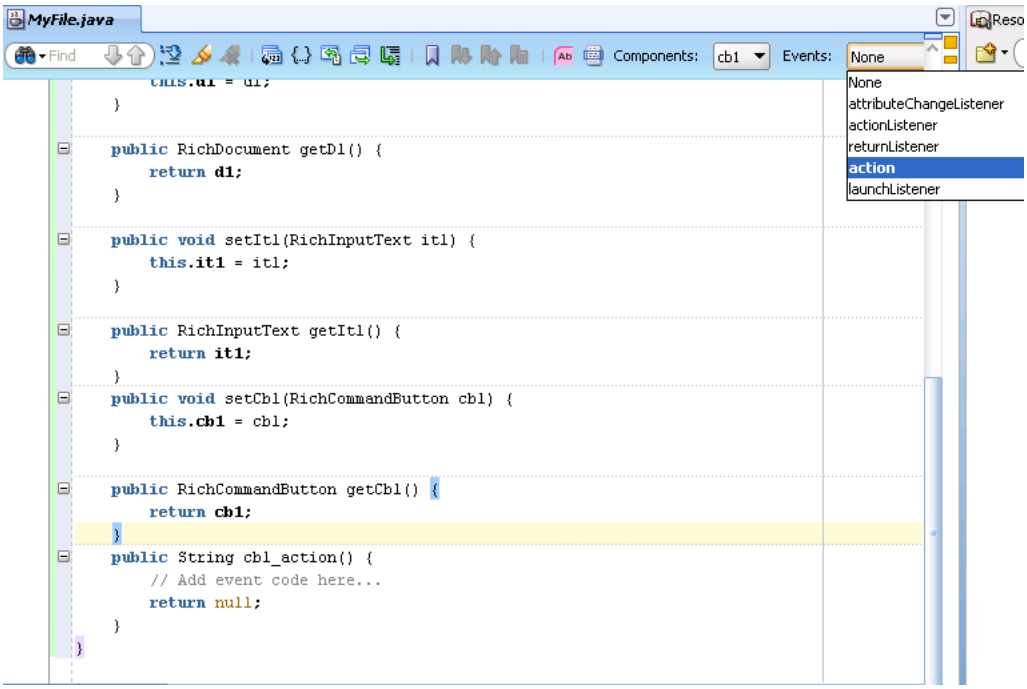
Example 2–11 shows the code added to the `faces-config.xml` file to register the page's backing bean as a managed bean.

Example 2–11 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>
```

In addition, when you edit a Java file that is a backing bean for a JSP, 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 2–7.

Figure 2–7 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 JSP 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 managed bean. If automatic binding is on and you change the managed bean selection, all existing bindings and new bindings are switched to the new bean.

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

2.4.4 How to Create a Facelets XHTML Page

You use the Create Facelets Page dialog to create the XHTML file.

To create an XHTML page:

1. In the Application Navigator, right-click the directory where you would like the page to be saved, and choose **New**. In the New Gallery, expand the **Web Tier** node, select **Facelets** and then **Facelets Page** and click **OK**.

   Tip: Click the All Technologies tab in the New Gallery if Facelets is not a listed technology.

2. Complete the Create Facelets Page dialog. For help, click **Help** in the dialog.

2.4.5 What Happens When You Create a JSF XHTML Page

When you use the Create Facelets Page dialog to create an XHTML page, JDeveloper creates the physical file and adds the code necessary to import the component libraries and display a page. Example 2–3 shows an `.xthml` page when it is first created by JDeveloper.

Example 2–12 Declarative Page Source Created by JDeveloper

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EE"
         "http://www.w3.org/TR/xhtml1/DTD/xhtall-transitional.dtd">
<f:view xmlns="http://www.w3.org/1999/xhtml"
        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">
  <af:document>
    <af:form/>
  </af:document>
</f:view>
```

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

- Facelets Runtime
- ADF Faces Runtime 11
- ADF Common Runtime
- ADF DVT Faces Runtime
- Oracle JEWT
- ADF DVT Faces Databinding Runtime

JDeveloper also adds entries to the `web.xml` file, as shown in Example 2–13.

Example 2–13 Code in the web.xml File After a JSF XHTML Page is Created

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<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">
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>
  <context-param>
    <param-name>org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION</param-name>
    <param-value>false</param-value>
  </context-param>
  <context-param>
    <param-name>oracle.adf.view.rich.versionString.HIDDEN</param-name>
    <param-value>false</param-value>
  </context-param>
  <context-param>
    <param-name>org.apache.myfaces.trinidad.FACELETS_VIEW_MAPPINGS</param-name>
    <param-value>*.xhtml</param-value>
  </context-param>
  <context-param>
```

```
  <param-name>facelets.SKIP_XML_INSTRUCTIONS</param-name>
  <param-value>true</param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.trinidad.ALTERNATE_VIEW_HANDLER</param-name>
  <param-value>
    org.apache.myfaces.trinidadinternal.facelets.TrinidadFaceletViewHandler
  </param-value>
</context-param>
<context-param>
  <param-name>facelets.DEVELOPMENT</param-name>
  <param-value>true</param-value>
</context-param>
<context-param>
  <param-name>facelets.SKIP_COMMENTS</param-name>
  <param-value>true</param-value>
</context-param>
<context-param>
  <param-name>facelets.DECORATORS</param-name>
  <param-value>
    oracle.adfinternal.view.faces.facelets.rich.AdfTagDecorator
  </param-value>
</context-param>
<context-param>
  <param-name>facelets.RESOURCE_RESOLVER</param-name>
  <param-value>
    oracle.adfinternal.view.faces.facelets.rich.AdfFaceletsResourceResolver
  </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>
</filter-mapping>
<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.adfinternal.view.faces.bi.renderkit.graph.GraphServlet
  </servlet-class>
</servlet>
<servlet>
  <servlet-name>BIGAUGESERVLET</servlet-name>
  <servlet-class>
    oracle.adfinternal.view.faces.bi.renderkit.gauge.GaugeServlet
```

```
    </servlet-class>
  </servlet>
  <servlet>
    <servlet-name>MapProxyServlet</servlet-name>
    <servlet-class>
      oracle.adfinternal.view.faces.bi.renderkit.geoMap.servlet.MapProxyServlet
    </servlet-class>
  </servlet>
  <servlet>
    <servlet-name>GatewayServlet</servlet-name>
    <servlet-class>
      oracle.adfinternal.view.faces.bi.renderkit.graph.FlashBridgeServlet
    </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>
  <servlet-mapping>
    <servlet-name>GatewayServlet</servlet-name>
    <url-pattern>/flashbridge/*</url-pattern>
  </servlet-mapping>
</web-app>
```

An entry is also created in the `faces-config.xml` file for the view handler, as shown in Example 2–14.

Example 2–14 Generated faces-config.xml File for an XHTML Page

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

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 2–15.

Example 2–15 Generated trinidad-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>fusion</skin-family>
</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.

2.4.6 How to Add ADF Faces Components to JSF Pages

Once you have created a page, you can use the Component Palette 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 III, "Using ADF Faces Components".

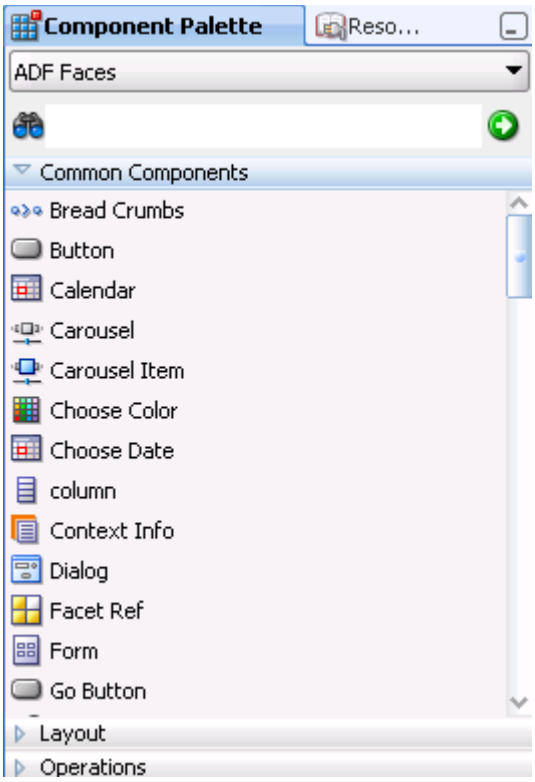
> **Note:** You cannot use ADF Faces components on the same page as MyFaces Trinidad components (`tr:` tags) or other AJAX-enabled library components. You can use Trinidad HTML tags (`trh:`) on the same page as ADF Faces components, however you may experience some browser layout issues. You should always 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.

To add ADF Faces components to a page:

1. In the Application Navigator, double click a JSF page to open it in the editor.
2. If the Component Palette is not displayed, from the menu choose **View > Component Palette**. By default, the Component Palette is displayed in the upper right-hand corner of JDeveloper.
3. In the Component Palette, use the dropdown menu to choose **ADF Faces**.

   The components are contained in three accordion panels: **Common Components**, **Layout**, and **Operations**. Figure 2–8 shows the Component Palette displaying the Common Components for ADF Faces.

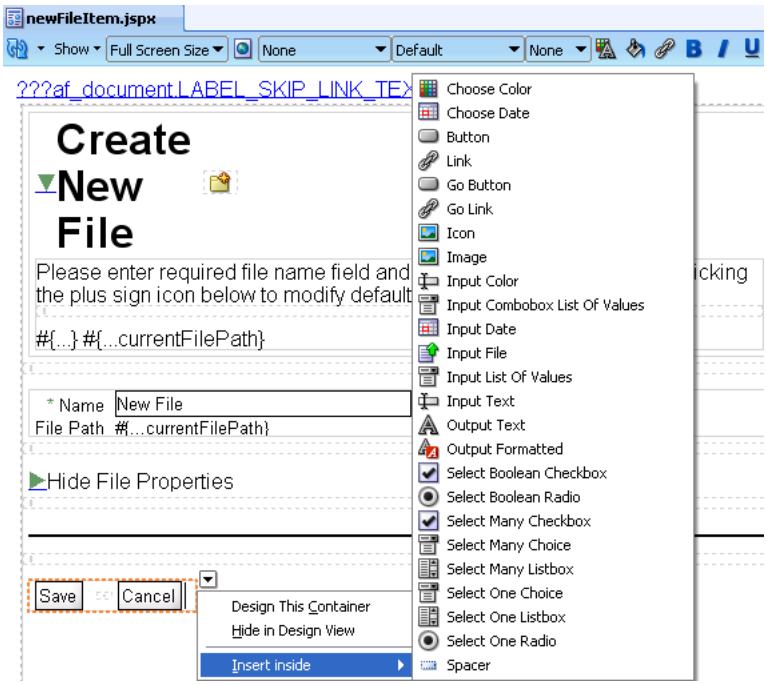
Figure 2–8 Component Palette in JDeveloper



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. Figure 2–9 shows a page in the visual editor.

Figure 2–9 Page Displayed in the Visual Editor



Tip: You can also drag and drop components from the palette 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.

2.4.7 What Happens When You Add Components to a Page

When you drag and drop components from the Component Palette 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 2–16 shows the code when you drop an Input Text and a Button component from the palette.

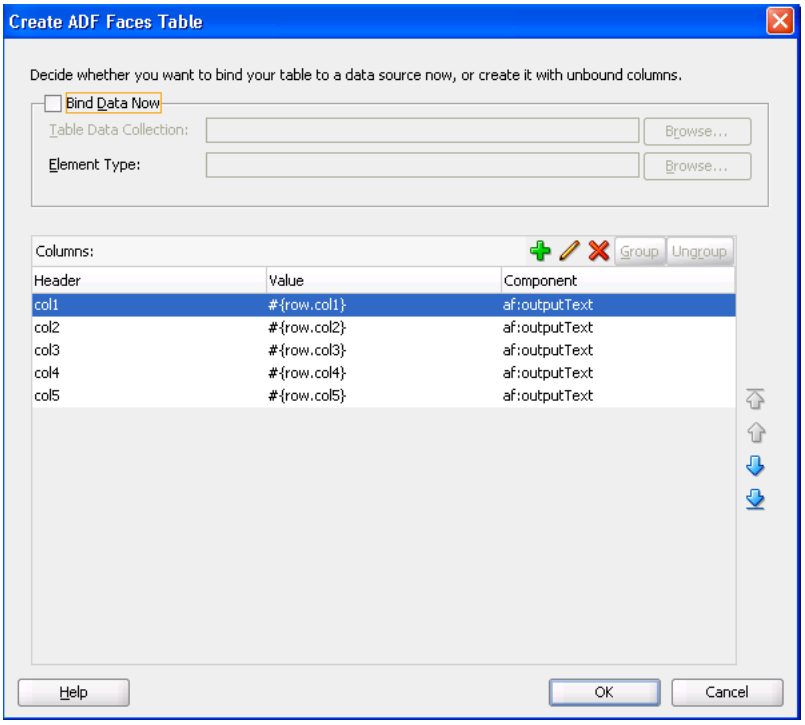
Example 2–16 JDeveloper Declaratively Adds Tags to a JSF Page

```
<af:inputText label="Label 1" id="it1"/>
<af:commandButton text="commandButton 1" id="cb"/>
```

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 2.4.3, "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 2–10 shows the Table wizard used to create a table component and the table's child column components.

Figure 2–10 Table Wizard in JDeveloper



Example 2–17 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 2–17 Declarative Code for a Table Component

```
<af:table var="row" 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>
```

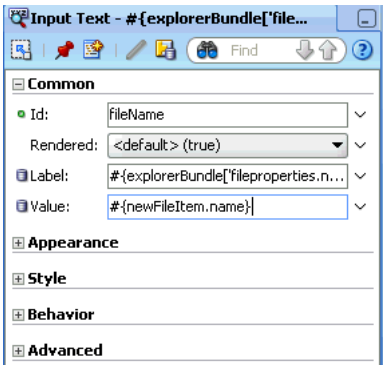
2.4.8 How to Set Component Attributes

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

> **Tip:** If the Property Inspector is not displayed, choose **View > Property Inspector** from the main menu.

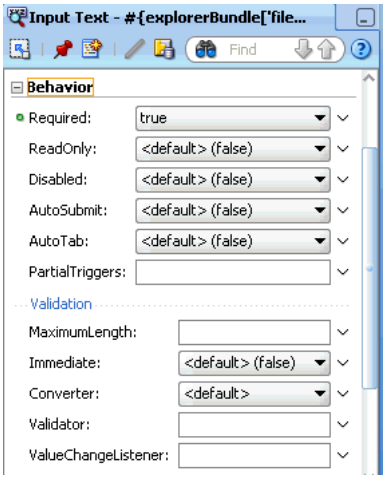
Figure 2–11 shows the Property Inspector displaying the attributes for an `inputText` component.

Figure 2–11 JDeveloper Property Inspector



The Property Inspector has sections that group similar properties together. For example, the Property Inspector 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 2–12 shows the Behavior section of the Property Inspector for an `inputText` component.

Figure 2–12 Behavior Section of the Property Inspector



To set component attributes:

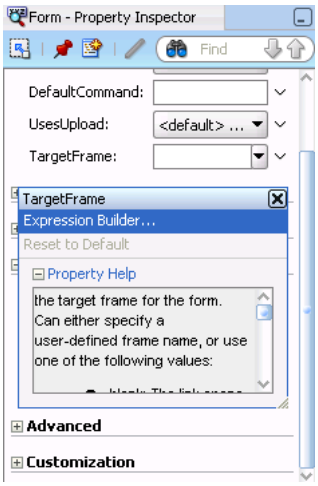
1. Select the component, in the visual editor, in the Structure window, or by selecting the tag directly in the source editor.
2. In the Property Inspector, expand the section that contains the attribute you wish to set.

   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 2.5, "Creating EL Expressions." This popup also displays a description of the property, as shown in Figure 2–13.

Figure 2–13 Property Tools and Help



2.4.9 What Happens When You Use the Property Inspector

When you use the Property Inspector 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.

2.5 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 2.6, "Creating and Using Managed Beans." For more information about EL expressions, see the Java EE 5 tutorial at http://www.oracle.com/technetwork/java/index.html.

2.5.1 How to Create an EL Expression

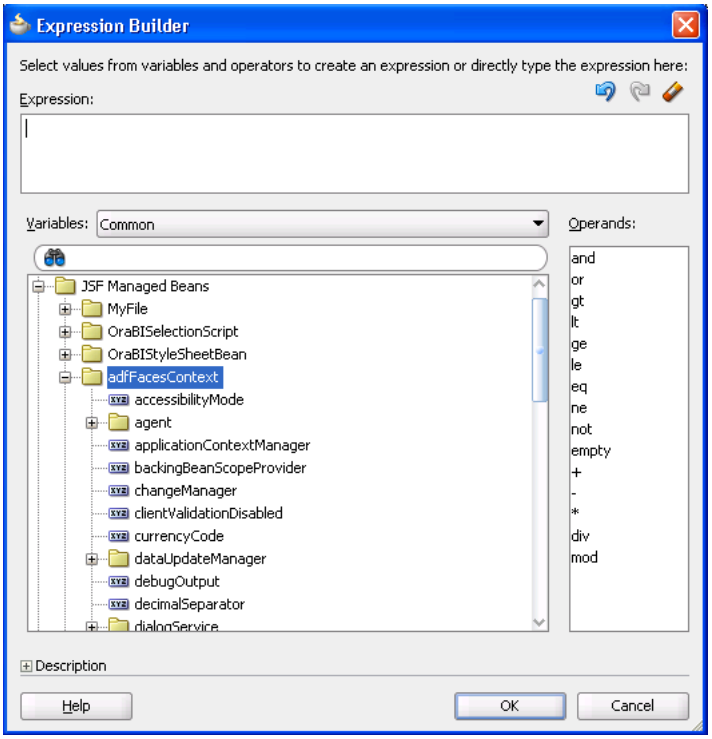
You can create EL expressions declaratively using the JDeveloper Expression Builder. You can access the builder from the Property Inspector.

To use the Expression Builder:

1. In the Property Inspector, locate the attribute you wish to modify and use the right most dropdown menu to choose **Expression Builder**.
2. Create expressions using the following features:
   - Use the **Variables** tree to select items that you want to include in the expression. The tree contains 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 2–14.

Figure 2–14 adfFacesContext Objects in the Expression Builder



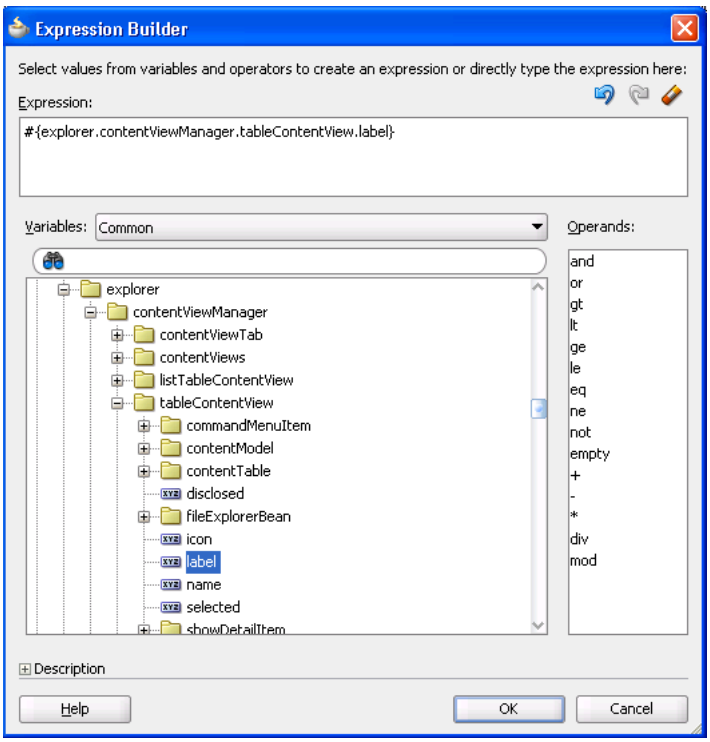
Tip: For more information about these objects, see the ADF Faces Javadoc.

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 2–15 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 2–15 The Expression Builder Dialog



2.5.2 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 2–18 shows how you can get a reference to an EL expression and return (or create) the matching object.

Example 2–18 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 2–19 shows how you can resolve a method expression.

Example 2–19 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 2–20 shows how you can set a new object on a managed bean.

Example 2–20 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);
  }
}
```

2.6 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 5 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

2.6.1 How to Create a Managed Bean in JDeveloper

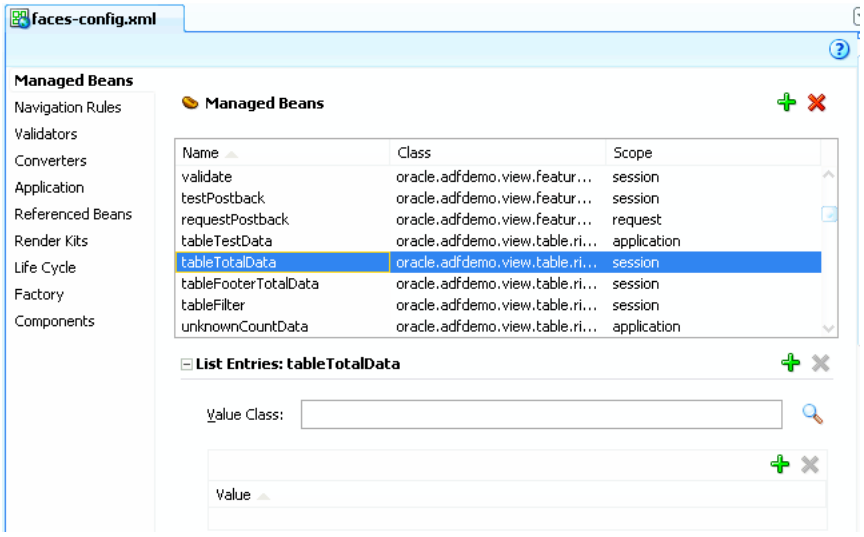
You can create a managed bean and register it with the JSF application at the same time using the overview editor for the `faces-config.xml` file.

To create and register a managed bean:

1. In the Application Navigator, open the `faces-config.xml` file.
2. In the editor window, click the **Overview** tab.
3. In the overview editor, click the **Managed Beans** tab.

   Figure 2–16 shows the editor for the `faces-config.xml` file used by the ADF Faces demo that contains the File Explorer application.

Figure 2–16 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 2.6.3, "What You May Need to Know About Component Bindings and Managed Beans."
- Use the `session` scope only for information that is relevant to the whole session, such as user or context information, or for high-availability reasons. Avoid using the `session` scope to pass values from one page to another.

For more information about the different object scopes, see Section 4.6, "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 Property Inspector, 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.

2.6.2 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 2–21 shows the code added to the `MyBean` class stored in the view package.

Example 2–21 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 2–22 shows the `managed-bean` element created for the `MyBean` class.

Example 2–22 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>
```

2.6.3 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 2–23 shows how a managed bean might use the `UIComponentReference` API to get and set values for a search field.

Example 2–23 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.

2.7 Viewing 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 either 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:

3 Using ADF Faces Architecture

This chapter outlines the major features of the ADF Faces client-side architecture.

This chapter includes the following sections:

3.1 Introduction to Using ADF Faces Architecture

The ADF Faces rich client framework (RCF) provides many of the features you need to create AJAX-type functionality in your web application, all built into the framework. A key aspect of the RCF is the sparsely populated client-side component model. 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.

The main reason client components exist is to provide an API contract for the framework and for developers. You can think of a client-side component as a simple property container with support for event handling. Because client components exist only to store state and provide an API, they have no direct interaction with the DOM (document object model) whatsoever. All DOM interaction goes through an intermediary called the *peer*. Most of the inner workings of the framework are hidden from you. Using JDeveloper in conjunction with ADF Faces, you can use many of the architectural features declaratively, without having to create any code.

For example, because RCF 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 3.4, "Instantiating Client-Side Components," explains how to do this declaratively. You use the Property Inspector 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 3.9, "Understanding Rendering and Visibility."

Other functionality may require you to use the ADF Faces JavaScript API. For example, Section 3.5, "Locating a Client Component on a Page," explains how to use the API to locate a specific client-side component, and Section 3.7, "Accessing Component Properties on the Client," documents how to access specific properties.

The following RCF features are more complex, and therefore have full chapters devoted to them:

- ADF Faces additions to the lifecycle: The ADF Faces framework extends the JSF lifecycle, providing additional functionality, including a client-side value lifecycle. For more information, see Chapter 4, "Using the JSF Lifecycle with ADF Faces."
- Event handling: ADF Faces adheres to standard JSF event handling techniques. In addition, the RCF provides AJAX-based rich postbacks (called *partial page rendering*), as well as a client-side event model. For more information, see Chapter 5, "Handling Events."
- Conversion and validation: ADF Faces input components have built-in capabilities to both convert and validate user entries. You can also create your own custom converters and validators. For more information, see Chapter 6, "Validating and Converting Input."
- Partial page rendering: Partial page rendering (PPR) allows small areas of a page to be refreshed without the need to redraw the entire page. Many ADF Faces components have built-in PPR functionality. In addition, you can declaratively configure PPR so that an action on one component causes a rerender of another. For more information, see Chapter 7, "Rerendering Partial Page Content."
- Geometry management: ADF Faces provides a number of layout components, many of which support geometry management by automatically stretching their contents to take up available space. For more information, see Chapter 8, "Organizing Content on Web Pages."
- Messaging and help: The RCF 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 17, "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 RCF 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 18, "Working with Navigation Components."
- Reusable components: The RCF provides three reusable building blocks that can be used by multiple pages in your application: page fragments that allow you to create a part of a page (for example an address input form); page templates that can provide a consistent look and feel throughout your application that can be updated with changes automatically propagating to all pages using the template; and declarative components that are composite components that developers can reuse, ensuring consistent behavior throughout the application. For more information, see Chapter 19, "Creating and Reusing Fragments, Page Templates, and Components."
- Applying skins: The RCF allows you to create your own look and feel by creating skins used by the ADF Faces components to change their appearance. For more

information, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

- Internationalization and localization: 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 21, "Internationalizing and Localizing Pages."
- Accessibility: ADF Faces components have built-in accessibility support for user agents, for example a web browser rendering to nonvisual media such as a screen reader or magnifier. Accessibility support also includes access keys that allow users to access components and links using only the keyboard, and audit rules that provide directions to create accessible images, tables, frames, forms, error messages, and popup dialogs using accessible HTML markup. For more information, see Chapter 22, "Developing Accessible ADF Faces Pages."
- User-driven personalization: Many ADF Faces components, such as the `panelSplitter`, 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 31, "Allowing User Customization on JSF Pages."
- Drag and drop capabilities: The RCF allows the user to move (cut and paste), copy (copy and paste), or link (copy and paste as a link) data from one location to another. When the drop is completed, the component accepting the drop rerenders using partial page rendering. For more information, see Chapter 32, "Adding Drag and Drop Functionality."

The remainder of this chapter focuses on working with the client-side framework.

3.2 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 using the `af:clientListener` tag.

For example, suppose you have a button that, when clicked, causes a "Hello World" alert to be displayed. You would first register a listener with the button that will invoke an event handler, as shown in Example 3–1.

Example 3–1 Registering a Client Listener

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

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

Next, implement the handler in a JavaScript function, as shown in Example 3–2.

Example 3–2 JavaScript Event Handler

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

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 5.3, "Using JavaScript for ADF Faces Client Events."

3.3 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.

3.3.1 How to Use Inline JavaScript

Create and use inline JavaScript in the same way you would in any JSF application. Once the JavaScript is on the page, use a `clientListener` tag to invoke it.

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 3–3.

Example 3–3 MyFaces Trinidad Tag Library on a Page

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          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. Create the JavaScript on the page.

   For example, the `sayHello` function shown in Example 3–2 might be included in a JSF page as shown in Example 3–4.

Example 3–4 Inline JavaScript

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

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

3. In the Structure window, right-click the component that will invoke the JavaScript, and choose **Insert inside *component*** > **ADF Faces** > **Client Listener**.
4. 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.

3.3.2 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.

To access a JavaScript library from a page:

1. Below the `document` tag, add the code shown in bold in Example 3–5 and replace `/mySourceDirectory` with the relative path to the directory that holds the JavaScript library.

Example 3–5 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 fully qualified name of the function. For example, if the `sayHello` function was in the `MyScripts` library, you would enter `MyScripts.sayHello`. In the **Type** field, select the event type that should invoke the function.

3.3.3 What You May Need to Know About Accessing Client Event Sources

Often when your JavaScript needs to access a client, 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 3–6 shows the `sayHello` function accessing the source client component in order to display its name.

Example 3–6 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 5.3, "Using JavaScript for ADF Faces Client Events." For more information about accessing client-side properties, see Section 3.7, "Accessing Component Properties on the Client." For a complete description of how client events are handled at runtime, see Section 5.3.6, "What Happens at Runtime: How Client-Side Events Work."

3.4 Instantiating Client-Side Components

The RCF does not make any guarantees about which components will have corresponding client-side component instances by default. You will usually interact with client-side components by registering a `clientListener` handler. When a component has a registered `clientListener` handler, it will automatically have client-side representation. If you have to access another component on the client, then explicitly configure that component to be available on the client by setting the `clientComponent` attribute to `true`.

> **Performance Tip:** Only set `clientComponent` to `true` if you plan on interacting with the component programmatically on the client.

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 3–7 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 3–7 Adding a Component

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

```
</af:commandButton>

<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.

3.5 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 3–8 shows the `sayHello` function finding the `outputText` component using the component's ID.

Example 3–8 Finding a Client Component Using findComponent()

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

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

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

Instead of using the `AdfUIComponent.findComponent(expr)` method, you can use the `AdfPage.PAGE.findComponentByAbsoluteId(absolute expr)` method when you know the absolute identifier for the component, but you don't have a component instance to call `AdfUIComponent.findComponent(expr)` on. `AdfPage.PAGE` is a global object that provides a static reference to the page's context object. However, if the component you are finding is within a naming container, then you must use `AdfUIComponent.findComponent`. For more information, see Section 3.5.1, "What You May Need to Know About Finding Components in Naming Containers."

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.

3.5.1 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 use the fully qualified JSF client ID (meaning, prefixed with the IDs of all `NamingContainer` components that contain the component) with a leading `NamingContainer.SEPARATOR_CHAR` character, for example:

```
":" + (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.

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). See the ADF Faces JavaScript documentation for more information.

3.6 Determining the User's Current Location

ADF Faces provides JavaScript APIs that return the current contextual page information, in response to an event. The `AdfPage.prototype.getViewId()` function returns the identifier for the currently displayed view. This ID is set when either a full page render or a partial page navigation occurs. The `AdfPage.prototype.getComponentsByType(componentType)` function returns an array of component instances that match the given component type.

For example, say your application contains a page with tabs, and each tab is made up of a number of regions. Each region could contain other nested regions as well. You can use the APIs to return a String identifier that is a combination of the `viewId` of the entire page and the `viewIds` of the fragments displayed in each of the regions currently rendered on the page, as shown in Example 3–9.

3.6.1 How to Determine the User's Current Location

In order to retrieve the viewID property of the region component on the client, the user activity monitoring feature needs to be enabled by setting a parameter in the `web.xml` file. You then create JavaScript code that builds a String representation of the `viewIds` that make up the current page.

To determine a context identifier:

1. Double-click the `web.xml` file.
2. In the source editor, set the `oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER` to true.

   This parameter notifies ADF Faces that the `ExecutionContextProvider` service provider is enabled. This service monitors and aggregates user activity information for the client-initiated requests.
3. Set `oracle.adf.view.rich.automation.ENABLED` to true.

   This parameter ensures that component IDs are set for all components. For more information, see Section A.2.3.11, "Test Automation."
4. Create the JavaScript to build the context identifier (for more information about adding JavaScript, see Section 3.3, "Adding JavaScript to a Page").

   Example 3–9 shows JavaScript used to get the current view ID for a region.

Example 3–9 JavaScript to Retrieve viewIds

```
/**
 * Returns a String identifier comprising the page viewId and viewIds
 * of the fragments displayed in each of the displayed regions
*/
TestLibrary.prototype.getCurrentPageInfo = function()
{
  var pageIdentifier = null;
  var page = AdfPage.PAGE;
  if (page)
  {
    // get the viewId of the page
    var viewId = page.getViewId();
    // get all region components currently displayed on the page
    var regionComponents = page.getComponentsByType("oracle.adf.RichRegion");
    var regionViewIds = new Array();
    for (var index = 0; index < regionComponents.length; index++)
```

```
    {
      var regionComp = regionComponents[index]);
      if (regionComp)
      {
        regionViewIds.push(regionComp.getProperty("viewId"));
      }
    }
    // construct page identifier
    if (viewId != null && regionViewIds.length > 0)
      contextId = viewId.concat(regionViewIds.toString());
  }
  return contextId;
}
```

3.7 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 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.

As noted in Section 1.2.2, "ADF Faces Architectural Features," 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 4.2, "Using the Immediate Attribute"). Consequently, the `immediate` property is a secured property.

Attempts to set any other secured property from JavaScript will fail. For more information, see Section 3.7.2, "How to Unsecure the disabled Property." Table 3–1 shows the secure properties on the client components.

Table 3–1 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`

`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`

Table 3–1 (Cont.) Secure Client Properties

Component	Secure Property
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
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

Table 3–1 (Cont.) Secure Client Properties

Component	Secure Property
AdfUIChart	chartDrillDownListener
AdfUIColumn	sortProperty
AdfUICommand	actionExpression returnListener launchListener immediate
AdfUIComponentRef	componentType
AdfUIEditableValueBase	immediate valid required localValueSet submittedValue requiredMessageDetail
AdfUIMessage.js	for
AdfUINavigationLevel	level
AdfUINavigationTree	rowDisclosureListener startLevel immediate
AdfUIPage	rowDisclosureListener immediate
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

Table 3–1 (Cont.) Secure Client Properties

Component	Secure Property
AdfUITreeBase	immediate
	rowDisclosureListener
	selectionListener
	focusRowKey
	focusListener
AdfUITreeTable	rowsByDepth
	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. When you set the unsecure property to true, the disabled property (and only the disabled property) will be made unsecure.

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.

3.7.1 How to Set Property Values on the Client

The RCF provides setXYZ convenience functions that provide calls to the AdfUIComponent setProperty() function. The setProperty() function takes the following arguments:

- Property name (required)
- New value (required)

3.7.2 How to Unsecure the disabled Property

You 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:commandButton text="commandButton 1" id="cb1" unsecure="disabled"/>
```

Once you set the unsecure attribute to disabled, a malicious JavaScript could change the disabled attribute unwittingly. 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.

If you did not set the `unsecured` attribute to `disabled`, the approval button would remain disabled until a round-trip to the server occurs, where logic determines if the current user can approve the expense. But because you want the button to display correctly as the page loads the expense, say you set the `unsecure` attribute to `disabled`. Now you can 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 this issue, you must ensure that your application still performs the same logic as if the round-trip to the server had happened. In the expense report approval screen, you might have JavaScript that checks that the amount is under $200, but you still need to have the action for the approval button 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.

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.

3.7.3 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 new values (as long as the value is different). Also, setting a property may cause the component to rerender itself.

3.8 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 RCF 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 5.4.3, "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 3.4, "Instantiating Client-Side Components."

3.8.1 How to Create Bonus Attributes

You can use the Component Palette to add a bonus attribute to a component.

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 Component Palette, from the Operations panel, drag and drop a **Client Attribute** as a child to the component.
3. In the Property Inspector, set the **Name** and **Value** attributes.

3.8.2 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 5.4, "Sending Custom Events from the Client to the Server."

3.9 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 RCF 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.

> **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 3–10 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 3–10 Rendered and Not Rendered Components

```
<af:panelGroupLayout layout="horizontal">
  <af:inputText label="Input some text" id="input"
```

```
                value="#{myBean.inputValue}"/>
  <af:commandButton 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 3–11 shows how you might achieve the same functionality as shown in Example 3–10, 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 3–11 Visible and Not Visible Components

```
<af:panelGroupLayout layout="horizontal">
  <af:inputText label="Input some text" id="input"
                value="#{myBean.inputValue}"/>
  <af:commandButton 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 3–12 shows the page code for JavaScript that handles the visiblity of the components.

Example 3–12 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)
    }

 }
```

3.9.1 How to Set Visibility Using JavaScript

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

To set visibility:

1. Create the JavaScript that can toggle the visibility. Example 3–12 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 Property Inspector and set the `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 3–13 shows the full page code used to toggle visibility with JavaScript.

Example 3–13 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:commandButton text="Enter" clientComponent="true">
        <af:clientListener method="showText" type="action"/>
      </af:commandButton>
    </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>
```

3.9.2 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 RCF 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

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:

4.1 Introduction to the JSF Lifecycle and ADF Faces

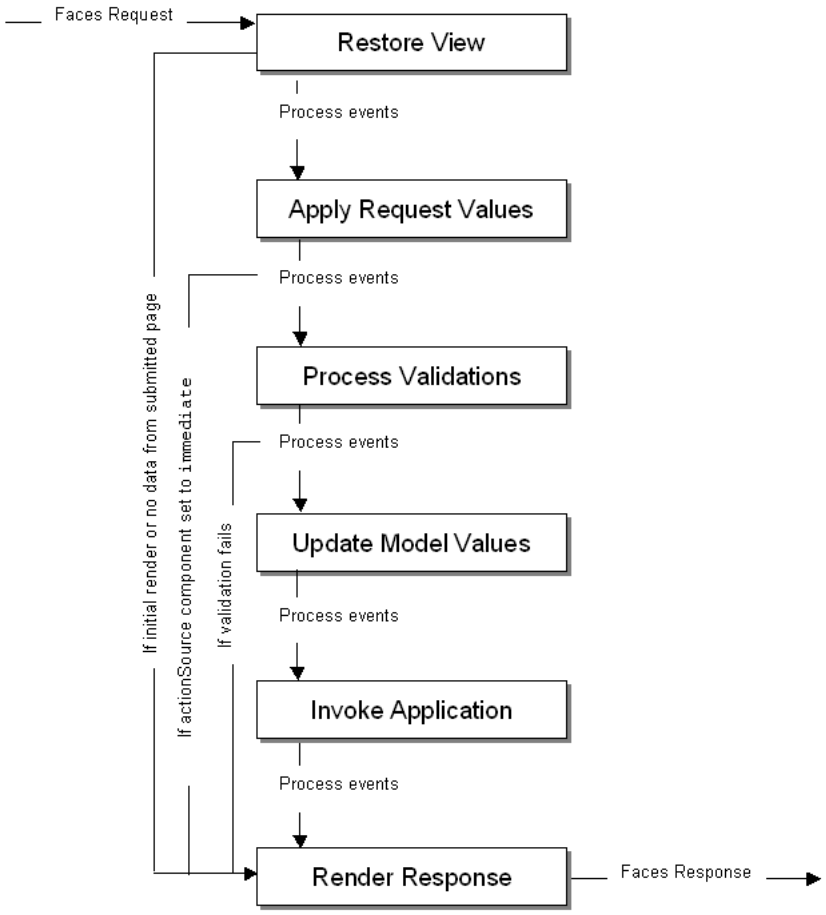
Because the ADF Faces rich client framework (RCF) extends the JSF framework, any application built using the ADF Faces rich client framework uses the standard JSF page request lifecycle. However, the ADF Faces framework extends that lifecycle, providing additional functionality, such as a client-side value lifecycle, a subform component that allows you to create independent submittable regions 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 RCF 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.oracle.com/technetwork/java/index.html`.

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 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. 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.

Figure 4–1 shows the JSF lifecycle of a page request. As shown, events are processed before and after each phase.

Figure 4–1 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 4.2, "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 6, "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 2010, and you had a `dateRestrictionValidator` validator that did not allow the user to pick Sundays. If the user entered `July 5, 2009` (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.

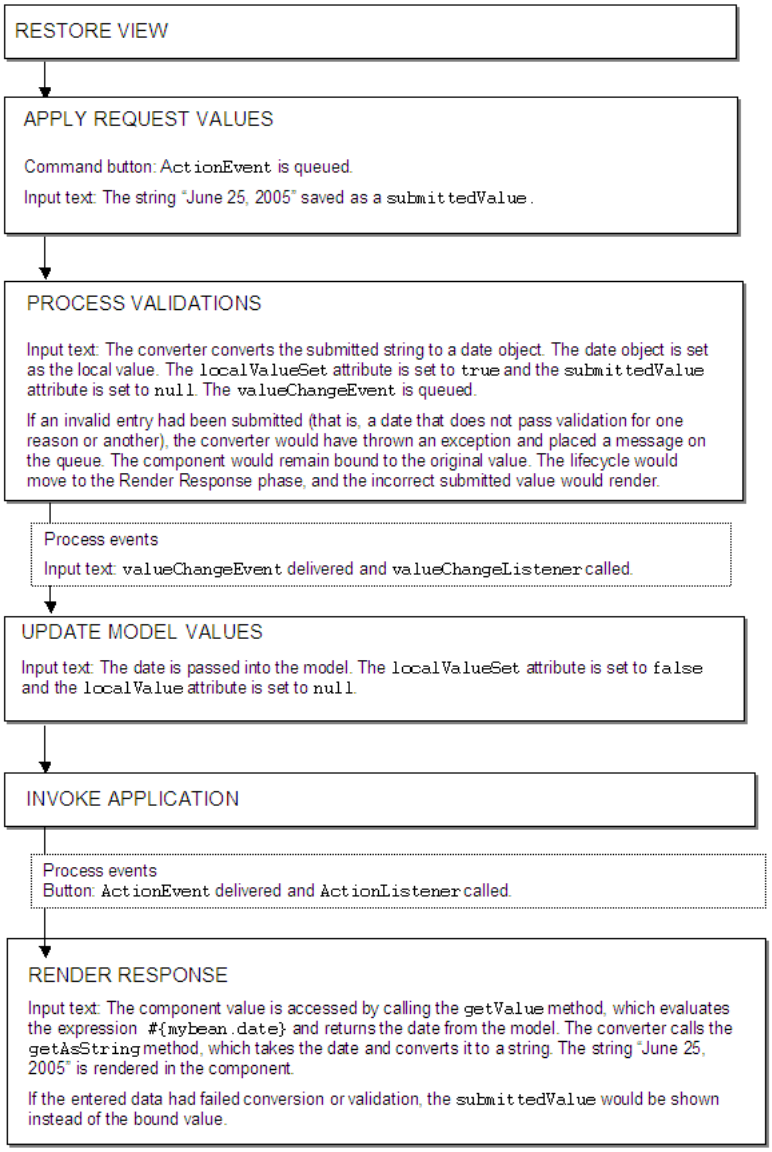
To help illustrate the lifecycle, consider a page that has a simple input text component where a user can enter a date and then click a command button to submit the entered value. A `valueChangeListener` method is also registered on the component. Example 4–1 shows the code for the example.

Example 4–1 Sample Code to Illustrate the JSF Lifecycle

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

Suppose a user enters the string "June 25, 2005" and clicks the submit button. Figure 4–2 shows how the values pass through the lifecycle and where the different events are processed.

Figure 4–2 Example of Values and Events in the JSF Lifecycle

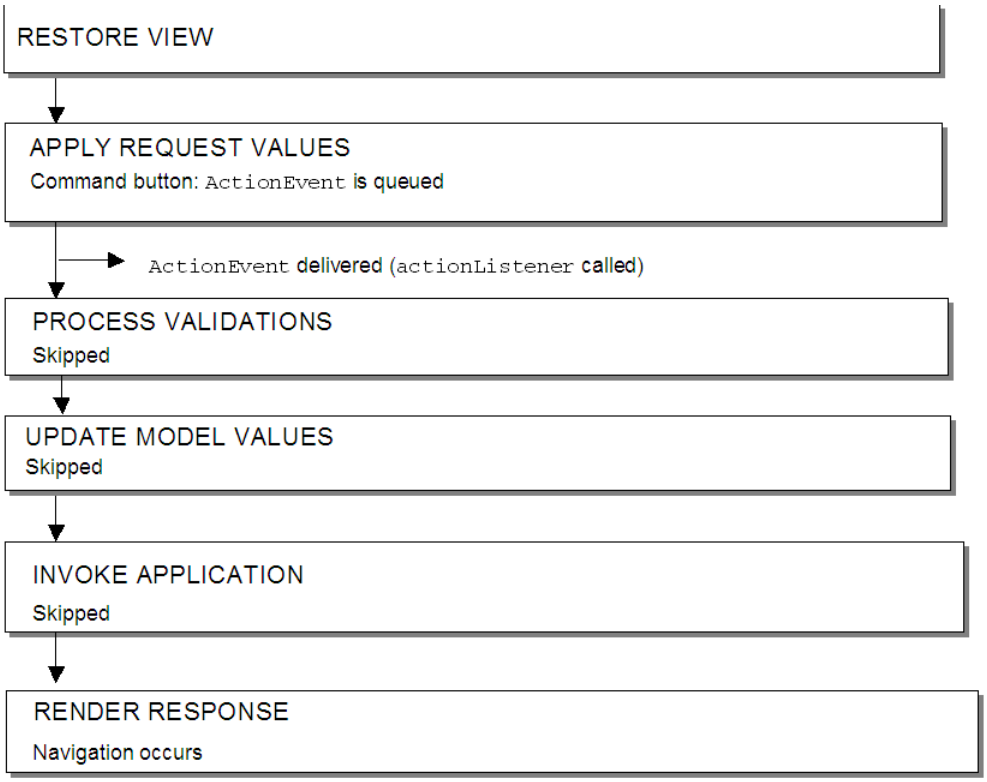


4.2 Using the Immediate Attribute

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 `commandButton`) are set to `immediate`, events are delivered in the Apply Request Values phase instead of in the Invoke Application phase. The `actionListener` handler then calls the Render Response phase, and the validation and model update phases are skipped.

For example, you might want to configure a Cancel button to be `immediate`, and have the action return a string used to navigate back to the previous page (for more information about navigation, see Chapter 18, "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 4–3.

Figure 4–3 Lifecycle for Command Button Set to Immediate

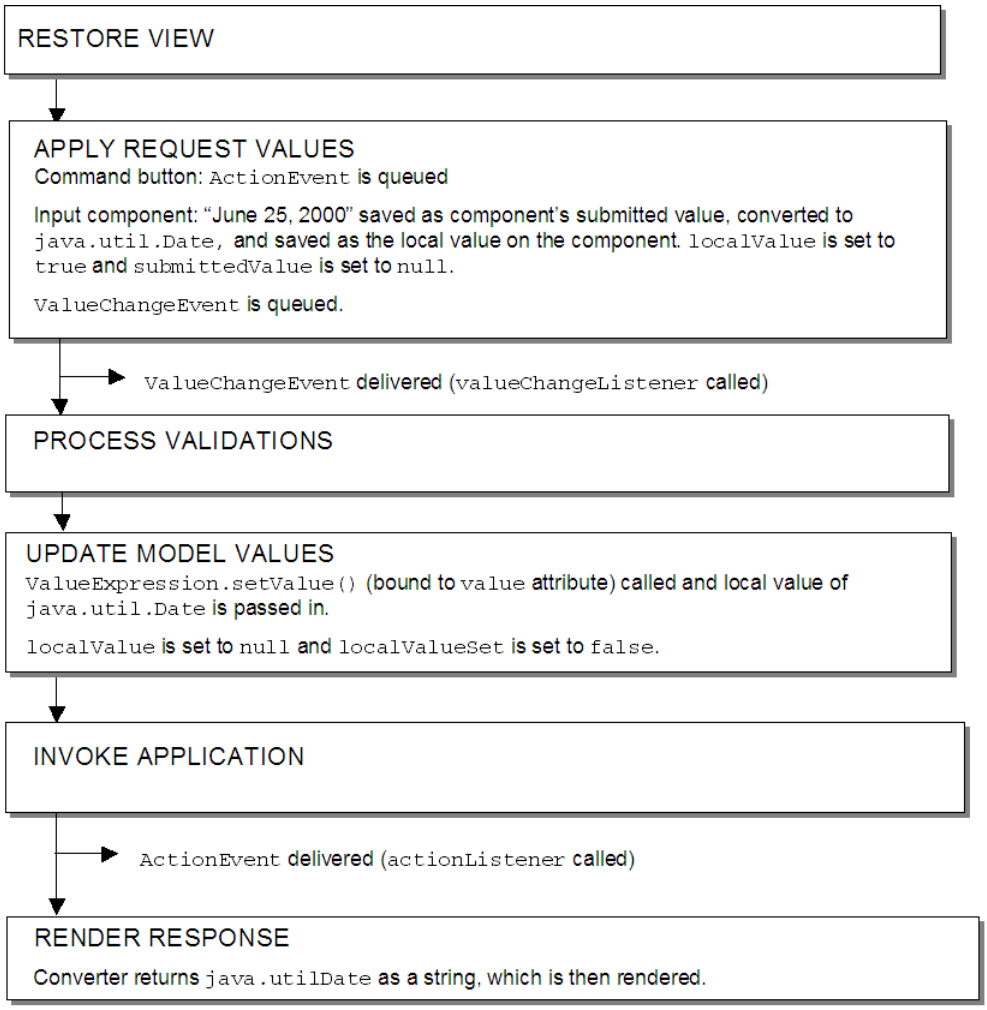


Note: A command button that does not provide any navigation and is set to `immediate` will also go directly to the Render Response phase: the Validation, Update Model, and Invoke Application phases are skipped, so any new values will not be pushed to the server.

As with command components, for components that invoke disclosure events, (such as a `showDetail` component), and for `editableValueHolder` components (components that hold values that can change, such as an `inputText` component) the events are 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 4–4 shows the lifecycle for an input component whose `immediate` attribute is set to `true`. The input component takes a date entered as a string and stores it as a date object when the command button is clicked.

Figure 4–4 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 18.8.1, "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 command button configured to invoke the search execution, and another input text component used to input a date with an associated command button used to submit the date. In this example, we 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 4–2 shows the code used for the two fields and two buttons.

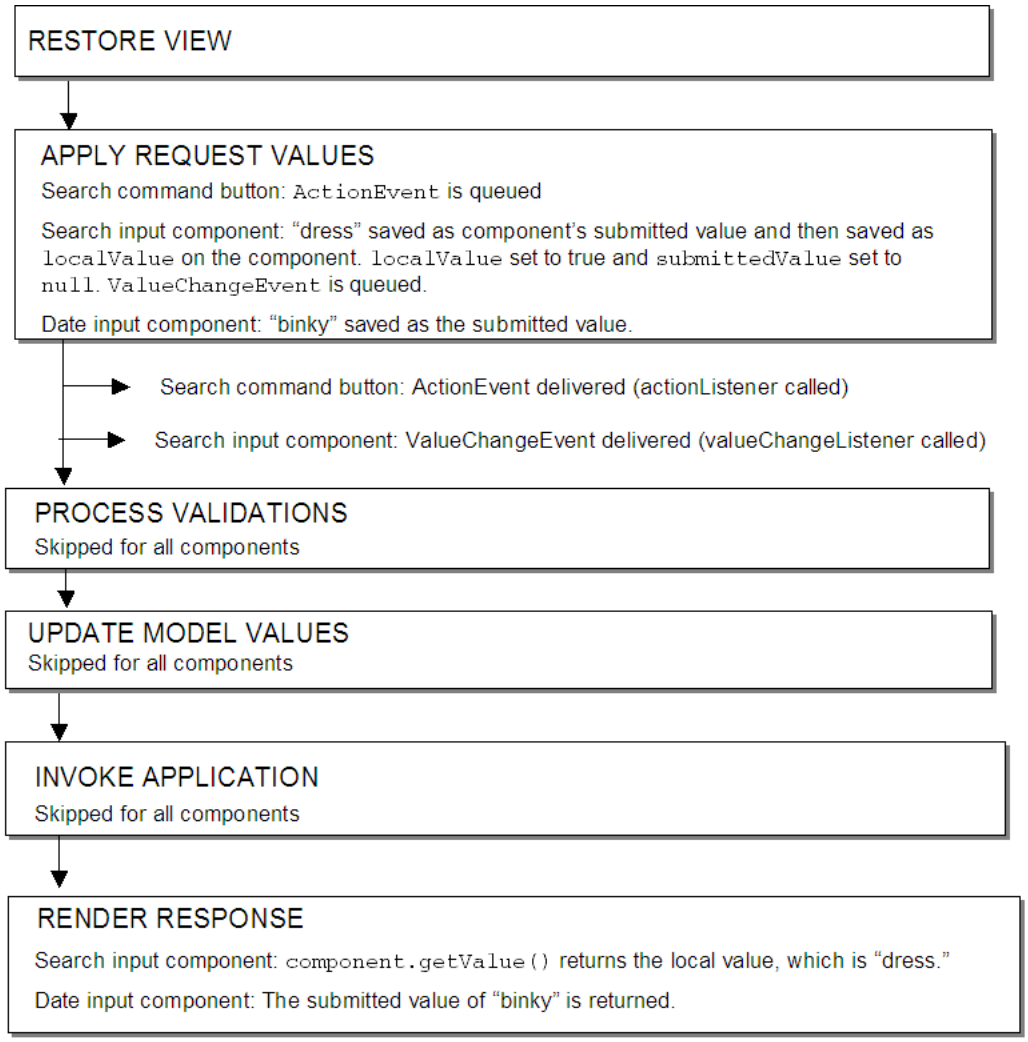
Example 4–2 Input Component and Command Components Using Immediate

```
<af:form>
  <af:inputText immediate="true" label="Search"  value="#{mybean.search}"
                valueChangeListener="#{mybean.searchValueChangeListener}"/>
  <af:commandButton 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:commandButton text="save"  actionListener="#{mybean.actionListener}"/>
</af:form>
```

Figure 4–5 shows the lifecycle for this page when a user does the following:

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

Figure 4–5 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 `editableValueHolder` component is marked as `immediate`, it will execute before the Update Model Values phase. This could be an issue when an immediate `actionSource` component requires data from an `editableValueHolder` component, as data entered into an `editableValueHolder` component is not 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.

To use the immediate attribute:

1. On the JSF page, select the component that you want to be immediate.

2. In the Property Inspector, expand the **Behavior** section and set the `immediate` attribute to `true`.

4.3 Using the Optimized Lifecycle

ADF Faces provides an optimized lifecycle that you can use when you want the JSF page request lifecycle (including conversion and validation) to be run only for certain components on a page. 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 4–6.

Figure 4–6 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 call the Render Response phase so that validation is not run on the input text component.

Instead of having to write this code in a listener, ADF Faces allows you to set boundaries on the page that allow the lifecycle to run just on components within the boundary. In order to determine the boundary, the framework must be notified of the root component to process. This component can be determined in two ways:

- Components: A region is an example of a component which the framework knows is a boundary. No matter what event is triggered inside a region, the lifecycle does not run on components outside the region.
- Events: Certain events indicate a component as a root. For example, the disclosure event sent when expanding or collapsing a `showDetail` component (see Section 8.8, "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. The lifecycle may also be run on any components configured to listen for that disclosure event. Configuring a component to listen for events on root components in order to be processed is called *cross-component refresh*.

Cross-component refresh allows you to set up dependencies so that the events from one component act as triggers for another component, known as the *target*. When any event occurs on the trigger component, the lifecycle is run on any target components, as well as on any child components of both the trigger and the target, causing only those components to be rerendered. This is considered a partial page rendering (PPR).

In the radio button example, 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 4–3.

Example 4–3 Example of Cross-Component Rendering

```
<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 `selectOneRadio` (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.

For more information about how the ADF Faces framework uses PPR, and how you can use PPR throughout your application, see Chapter 7, "Rerendering Partial Page Content."

4.3.1 What You May Need to Know About Using the Immediate Attribute and the Optimized Lifecycle

There may be cases where PPR will not be able to keep certain components from being validated. For example, suppose instead of using an `outputText` component, you want to use an `inputText` component whose `required` attribute is set to `true`, inside the `panelGroupLayout` component, as shown in Example 4–4.

Example 4–4 inputText Component Within a panelGroup Component Will Be Validated with Cross-Component PPR

```
<af:form>
  <af:selectBooleanRadio id="show2" autoSubmit="true" text="Show"
                         value="#{validate.show2}"/>
  <af:selectBooleanRadio id="hide2" autoSubmit="true" text="Hide"
                         value="#{validate.hide2}"/>
  <af:panelGroupLayout partialTriggers="show2 hide2">
    <af:inputText label="Required Field" required="true"
                                         rendered="#{validate.show2}"/>
  </af:panelGroupLayout>
</af:form>
```

In this example, the `inputText` component will be validated because the lifecycle runs on the root (the `selectOneRadio` 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 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 skip validation using the `immediate` attribute on the radio buttons. Doing so causes the `valueChangeEvent` on the buttons to run before the Process Validation phase of the `inputText` component. Then you need to add a `valueChangeListener` handler method that would call the Render Response phase

(thereby skipping validation of the input component), and set the values on the radio buttons and input component. Example 4–5 shows the JSF code to do this.

Example 4–5 Using the immediate Attribute and a valueChangeListener

```
<af:form>
  <af:selectOneRadio immediate="true"
                        valueChangeListener="#{validate.toggle}"
                      id="show2" autoSubmit="true" text="Show"
                        value="#{validate.show2}"/>
  <af:selectOneRadio id="hide2" autoSubmit="true" text="Hide"
                        value="#{validate.hide2}"/>
  <af:panelGroupLayout partialTriggers="show2 hide2">
    <af:inputText label="Required Field" required="true"
                  rendered="#{validate.show2}"/>
  </af:panelGroupLayout>
</af:form>
```

Example 4–6 shows the `valueChangeListener` code.

Example 4–6 valueChangeListener Sets the Value and Calls Render Response

```
public void toggle(ValueChangeEvent vce)
  {
    setShow2(Boolean.TRUE.equals(vce.getNewValue()));
    FacesContext.getCurrentInstance().renderResponse();
  }
```

4.3.2 What You May Need to Know About Using an LOV Component and the Optimized Lifecycle

For the `inputListOfValues` and `inputComboBoxListOfValues` components, the procedures described in Section 4.3.1, "What You May Need to Know About Using the Immediate Attribute and the Optimized Lifecycle," will not work. Consider the following example.

Suppose you have an `inputListOfValues` component from which a user selects an employee name, and an `inputText` component whose required attribute is set to `true`, which is updated with the employee's ID number once the employee is selected, as shown in Figure 4–7.

Figure 4–7 LOV component Updates the Input Component

To achieve this, you might set the **Empno** field to have the **Ename** field as a partial trigger, as shown in Example 4–7.

Example 4–7

```
<af:inputListOfValues label="Ename" id="lov0"
                      value="#{validateLOV.ename}" autoSubmit="true"
                      immediate="true"
                      popupTitle="Search and Select: Ename"
                      searchDesc="Choose a name"
                      model="#{validateLOV.listOfValuesModel}"
```

```
                    valueChangeListener="#{validateLOV.immediateValueChange}"
                    validator="#{validateLOV.validate}"/>
<af:inputText label="Empno" value="#{validateLOV.empno}" required="true"
              id="lovDependent01" partialTriggers="lov0"/>
```

As with the radio button and input component example in Section 4.3.1, "What You May Need to Know About Using the Immediate Attribute and the Optimized Lifecycle," once 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, so an error will be thrown, as shown in Figure 4–8.

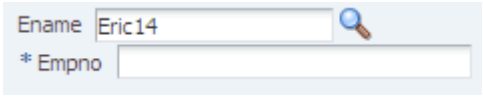
Figure 4–8 Validation Error is Thrown Because a Value is Required

However, the solution recommended in Section 4.3.1, "What You May Need to Know About Using the Immediate Attribute and the Optimized Lifecycle," of setting the LOV component's immediate attribute to `true` and using a `ValueChange`Listener on the LOV will not fix the validation error. For LOV components, the `ValueChangeEvent` is queued only when the value of the LOV component changes. For this reason, setting the `immediate` attribute to `true` has no effect when the user clicks the search icon, because at that point the ADF `LaunchPopupEvent` is queued for the Invoke Application phase always, regardless of the value of the `immediate` attribute. In other words, the optimized lifecycle is run as normal on both the root and target components and therefore the input component throws a validation error.

When the user selects a *new* value from the LOV popup, the LOV component queues two events. One is a `ValueChangeEvent` to signal a change in value for the component. The second is a `ReturnPopupEvent` queued for the Invoke Application phase, which gives application methods a chance to handle the selection. Both these events need to occur in order for the LOV to behave as expected.

As mentioned, the LOV component queues a `ValueChangeEvent` only when the user selects a new value. If you were to set the `immediate` attribute to `true` on the LOV component, this event would be queued for the Apply Request Values phase and the new value would be validated. In addition if you were to create a `ValueChangeListener` method for the LOV component, and in its implementation jump to the Render Response phase to avoid validation of the input component, the selected value would never get pushed to the model, the `ReturnPopupListener` would never get called during the Invoke Application phase, and the target input component would not get updated with new value, as shown in Figure 4–9.

Figure 4–9 Model is Not Updated



To resolve this issue of needing both the `ValueChangeEvent` and the `ReturnPopupEvent` to be queued as part of the same request and to have any target fields refreshed with newly selected values, instead of declaratively setting the LOV

component as a partial trigger for the input component and creating a method for the `ValueChangeListener`, you need to create a listener for the `ReturnPopupEvent`. This listener must programmatically set the input components as partial targets for the LOV. You do not need to set the LOV's `immediate` attribute to `true` because the input component is no longer a target for the LOV until the `ReturnPopupListener` method is executed, and so it will not fail validation because the lifecycle will not be run on it. And because a listener method is used for the `ReturnPopupEvent` instead of for the `ValueChangeEvent`, both events can be queued and the model updated appropriately.

Example 4–8 shows the needed page code for the LOV and input components.

Example 4–8

```
<af:inputListOfValues label="Ename" id="lov1"
                      value="#{validateLOV.ename}" autoSubmit="true"
                      returnPopupListener="#{validate.processReturnPopup}"
                      Title="Search and Select: Ename" searchDesc="Choose a name"
                      model="#{validateLOV.listOfValuesModel}"
                      validator="#{validateLOV.validate}"/>
 <af:inputText label="Empno" value="#{validateLOV.empno}" required="true"
               id="lovDependent1" binding="#{validate.lovDependent1}"/>
```

The input component uses its `binding` attribute to store the instance on a backing bean, allowing the instance to be accessed by the listener method. The listener method then accesses the input component and sets it as a partial target for the LOV, as shown in Example 4–9.

Example 4–9

```
AdfFacesContext.getCurrentInstance().addPartialTarget(_lovDependent1)
```

For more information about programmatically setting partial page rendering, see Section 7.3, "Enabling Partial Page Rendering Programmatically."

4.4 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 4.5, "Using Subforms to Create Regions 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 command 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 is 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.

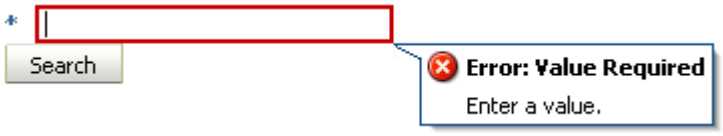
For example, ADF Faces provides the `required` attribute for input components, and this validation runs on the client. When you set this attribute to `true`, the framework will show an error on the page if the value of the component is `null`, without requiring a trip to the server. Example 4–10 shows code that has an `inputText` component's `required` attribute set to `true`, and a command button whose `actionListener` attribute is bound to a method on a managed bean.

Example 4–10 Simple Client-Side Validation Example

```
<af:form>
  <af:inputText id="input1" required="true" value="a"/>
  <af:commandButton 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 4–10. There will be no trip to the server; this error detection and message generation is all done on the client.

Figure 4–10 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 6, "Validating and Converting Input."

4.5 Using Subforms to Create Regions on a Page

In the JSF reference implementation, if you want to independently submit a region 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 region 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 9.2, "Defining Forms."

4.6 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 three 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.
- `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:

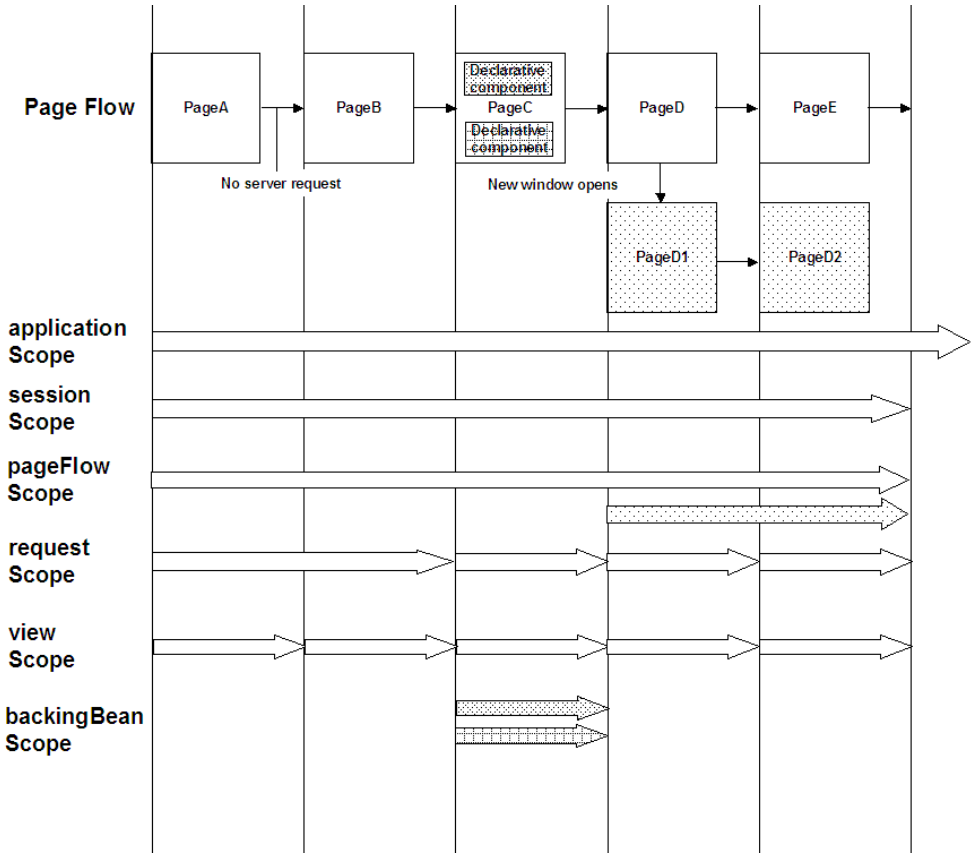
- `pageFlowScope`: The object is available as long as the user continues navigating from one page to another. If the user opens a new browser window and begins navigating, that series of windows will have its own `pageFlowScope` scope.
- `backingBeanScope`: Used for managed beans for page fragments and 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.

> **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 4–11 shows the time period in which each type of scope is valid, and its relationship with the page flow.

Figure 4–11 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4.7 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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:commandButton 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, `commandButton`) 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 `commandButton` 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.

4.7.1 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.

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();
```

4.7.2 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 Component Palette.
2. In the Component Palette, from 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}`.

4.7.3 What Happens at Runtime: Passing Values

When a user clicks a command 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.

5 Handling Events

This chapter describes how to handle events on the server as well as on the client.

This chapter includes the following sections:

5.1 Introduction to 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 user 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 appropriate 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

5.1.1 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 4.3, "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 rerendered, and not the whole page.

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 8.8, "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 5–1 shows the event types in ADF Faces, and whether or not the source component is an event root.

Table 5–1 Events and Event Root Components

Event Type	Component Trigger	Is Event Root
`action`	All command components	`false`
`dialog`	`dialog`	`false`
`disclosure`	`showDetail`, `showDetailHeader`	`true`
`disclosure`	`showDetailItem`	`true`
`focus`	`tree`, `treeTable`	`true`
`launch`	All command components	NA
`launchPopup`	`inputListOfValues`, `inputComboboxListOfValues`	`true`
`load`	`document`	NA
`poll`	`poll`	`true`
`popupOpened`	`popup`	NA
`popupOpening`	`popup`	NA
`popupClosed`	`popup`	NA

Table 5–1 (Cont.) Events and Event Root Components

Event Type	Component Trigger	Is Event Root
`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`	`true`
`selection`	`tree`, `treeTable`, `table`	`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 must set the `partialTrigger` attribute on those components to the ID of the event root component.

5.1.2 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 RCF 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.

5.2 Using ADF Faces Server Events

ADF Faces provides a number of server-side events. Table 5–2 lists the events generated by ADF Faces components on the server, and the components that trigger them.

Table 5–2 ADF Faces Server Events

Event	Triggered by Component...
ActionEvent	All command components
DialogEvent	dialog
DisclosureEvent	showDetail, showDetailHeader, showDetailItem
FocusEvent *	tree, treeTable
LaunchEvent	All command components
LaunchPopupEvent	inputListOfValues, inputComboboxListOfValues
LoadEvent **	document
PollEvent	poll
QueryEvent	query, quickQuery
QueryOperationEvent	query, quickQuery
RangeChangeEvent	table
RegionNavigationEvent	region
ReturnEvent	All command components
ReturnPopupEvent	inputListOfValues, inputComboboxListOfValues
RowDisclosureEvent	tree, treeTable
SelectionEvent	tree, treeTable, table
SortEvent	treeTable, table
ValueChangeEvent	All input and select components (components that implement EditableValueHolder)

* 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).

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.

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 5–1 shows the code for the tableSelectFileItem handler. (For information about creating and using managed beans, see Section 2.6, "Creating and Using Managed Beans.")

Example 5–1 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 Property Inspector, 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 5–2 shows sample code for registering a selection event listener method on a `table` component.

Example 5–2 Registering an Event Listener Method

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

5.3 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 5.3.5, "How to Prevent Events from Propagating to the Server."

Table 5–3 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 5–3 ADF Faces Client Events

Event Type	Event Class	Propagates to Server	Can Be Canceled	Triggered by Component
`action`	`AdfActionEvent`	Yes	Yes	All command components
`dialog`	`AdfDialogEvent`	Yes	Yes	`dialog` When user selects the OK or Cancel button in a dialog
`disclosure`	`AdfDisclosureEvent`	Yes	Yes	`showDetail`, `showDetailHeader`, `showDetailItem` When the disclosure state is toggled by the user
	`AdfFocusEvent`	Yes	Yes	`tree`, `treeTable`
	`AdfLaunchPopupEvent`	Yes	Yes	`inputListOfValues`, `inputComboboxListOfValues`
`inlineFrameLoad`	`AdfDomComponentEvent`	Yes	Yes	`inlineFrame` When the internal `iframe` fires its `load` event.
`load`	`AdfComponentEvent`	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.
	`AdfPollEvent`	Yes	Yes	`poll`
`popupOpened`	`AdfPopupOpenedEvent`	No	No	`popup` After a popup window or dialog is opened
`popupOpening`	`AdfPopupOpeningEvent`	No	Yes	`popup` Prior to opening a popup window or dialog
`popupClosed`	`AdfPopupClosedEvent`	No	No	`popup` After a popup window or dialog is closed
`propertyChange`	`AdfPropertyChangeEvent`	No	No	All components
`query`	`AdfQueryEvent`	Yes	Yes	`query`, `quickQuery` Upon a query action (that is, when the user clicks the search icon or search button)

Table 5–3 (Cont.) ADF Faces Client Events

Event Type	Event Class	Propagates to Server	Can Be Canceled	Triggered by Component
queryOperation	AdfQueryOperationEvent	Yes	Yes	query, quickQuery
	AdfReturnEvent	Yes	Yes	All command components
	AdfReturnPopupDataEvent	Yes	Yes	inputListOfValues, inputComboboxListOfValues
	AdfReturnPopupEvent	Yes	Yes	inputListOfValues, inputComboboxListOfValues
rowDisclosure	AdfRowDisclosureEvent	Yes	Yes	tree, treeTable When the row disclosure state is toggled
selection	AdfSelectionEvent	Yes	Yes	tree, treeTable, table When the selection state changes
sort	AdfSortEvent	Yes	Yes	treeTable, table When the user sorts the table data
touchStart touchMove touchEnd touchCancel	AdfComponentTouchEvent	No	Yes	All
valueChange	AdfValueChangeEvent	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 5–4

Table 5–4 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 5–3 shows an example of a JavaScript function associated with an action event.

Example 5–3 clientListener Tag

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

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

5.3.1 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.

To use client-side events:

1. Create the JavaScript event handler function. For information about creating JavaScript, see Section 3.3, "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 3.5, "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 5.3.2, "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 5.3.3, "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 5–3. If you do not need this extra processing, then you can cancel that propagation. For more information, see Section 5.3.5, "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`)

   1. Select the component to invoke the JavaScript, and in the Property Inspector, set **ClientComponent** to **true**.
   2. In the Component Palette, from the Operations panel, drag a **Client Listener** and drop it as a child to the selected component.
   3. 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 5–3 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 5–4 lists the keyboard and mouse event types.

      Example 5–4 shows the code used to invoke the `showHelpFileExplorerPopup` function from the `Explorer.js` JavaScript file.

Example 5–4 clientListener Tags on JSF Page

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

4. Add any attributes required by the function by dragging a **Client Attribute** from the Operations panel of the Component Palette, and dropping it as a child to the selected component. Enter the name and value for the attribute in the Property Inspector. Example 5–5 shows the code used to set attribute values for the `showAboutFileExplorerPopup` function.

Example 5–5 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>
```

Note: If you use the `attribute` tag (instead of the `clientAttribute` tag) to add application-specific attributes or bonus attributes to a server component, those attributes are not included on the client component equivalent. You can use the `clientAttribute` tag on the JSF page, and the value will then be available on both the server and client. For information about posting client values back to the server, see Section 5.4, "Sending Custom Events from the Client to the Server." For information about bonus attributes, see Section 3.8, "Using Bonus Attributes for Client-Side Components."

5.3.2 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 5–6, 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 5.3.3, "How to Use Client-Side Attributes for an Event").

Example 5–6 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.

5.3.3 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 5–7, 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 5–7 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 5–8.

Example 5–8 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.

5.3.4 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 5–9 shows how you can use `preventUserInput` in your JavaScript.

Example 5–9 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);
}
```

5.3.5 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 `commandButton` 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 5–10 shows the `showAboutFileExplorerPopup` function, which cancels its propagation.

Example 5–10 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 5–3). 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.

5.3.6 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 rich client 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 3, "Using ADF Faces 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.

5.3.7 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 5–11. The `getSource()` method accesses the `AdfUIComponent` class, which can then be used to find the component.

Example 5–11 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 3.5, "Locating a Client Component on a Page."

5.4 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 5–12 shows, this happens because the `inputText` field contains a `clientListener` that invokes a JavaScript function when the **Enter** key is pressed.

Example 5–12 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.

5.4.1 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.

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 5–13.

Example 5–13 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 5–14 shows the code used in the `SearchNavigatorView` managed bean that simply calls another method to execute the search and then refreshes the navigator.

Example 5–14 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 5.4.3, "What You May Need to Know About Marshalling and Unmarshalling Data."

3. Register the `clientListener` by dragging a **Client Listener** from the Operations panel of the Component Palette, and dropping 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. Register the server listener by dragging a **Server Listener** from the Operations panel of the Component Palette, and dropping 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 Property Inspector, for the `method` attribute, enter an expression that resolves to the method created in Step 2.

5.4.2 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.

5.4.3 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. When the client is browser-based, the two common strategies for marshalling are

JavaScript Object Notation (JSON) and XML. ADF Faces uses a mix of both of these strategies, with the information sent from the server to the client mostly as JSON and information sent from the client to the server as XML (for more information about JSON, see http://www.json.org).

When you send information from JavaScript to Java, the JavaScript data objects are converted (marshalled) into XML, which is then parsed back or unmarshalled into Java objects at the server-side. For example, consider a JSF page that has a `commandButton` component whose ID is `cmd`. When a user clicks the `commandButton` component, the client must communicate to the server that an `actionEvent` has been fired by this specific `commandButton`. In the `requestParameter` map, the information is mapped with the key using the format `event + . + id` where `id` is the ID of the component. So the `requestParameter` map key for the `commandComponent` would be the XML string stored as the value of the key `event.cmd`.

The XML fragment after marshalling in this example would be:

```
<m xmlns="http:/oracle.com/richClient/comm"><k v="type"><s>action</s></k></m>
```

The `m` in the example means that this should be unmarshalled into a map. The `k` denotes the key and the value is of type `String`. On the server side, this XML fragment is parsed into a `java.util.Map` of one entry having `type` (`java.lang.String`) as the key and `action` (`java.lang.String`) as the value.

The unmarshalled information is grouped per client ID, stored in the request map, and used when the components are being decoded. So in this example, when the `commandButton` is decoded, it will check for the presence of any client events using its client ID (`event.cmd`) and then queue an action event if one is found (the decode behavior is implemented in the renderer hierarchy for `commandButton` component).

Table 5–5 shows the mapping between corresponding JavaScript and Java types.

Table 5–5 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`

Marshalling from Java to JavaScript happens mostly through JSON. This type of marshalling is straightforward as JSON is the object literal notation in JavaScript. The client-components usually have their properties encoded in JSON. Consider the following example:

```
new AdfRichCommandButton('demoTemplate:richComand'
    {'partialSubmit':true,'useWindow':false})
```

The second argument ({'partialSubmit':true,'useWindow':false}) is a JSON object. There is no additional unmarshalling step required at the browser end as JSON can directly be parsed into the JavaScript environment as an object.

Encoding for a table also uses JSON to pass push messages to the client. The following is an example of an envelope containing a single encoded push message:

```
[{'rKey':'0','type':'update','data':[{'val':'Active Data Every Second: on row
0:78','prop':'value','cInd':0},{'val':'Active Data Every Second: on row
0:78','prop':'value','cInd':1}]}]
```

The envelope is a JavaScript `Array` with only one object, which describes the message. This message contains information about the type of change, the actual value of the data, and so on, that is then used by the client-side table peer to update the table itself.

Table 5–6 shows the mapping between corresponding Java and JavaScript types.

Table 5–6 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`
`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`.

5.5 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 5–15 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 5–15 Invoking a Method to Add JavaScript to a Response

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

Example 5–16 shows the `handleUpOneFolder` method that uses the `ExtendedRenderKitService` class.

Example 5–16 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();
    }

  }
```

5.6 Using 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 8.7.1, "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 9.3.2, "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 9.8.2, "How to Add the Ability to Insert Text into a richTextEditor Component."
- `showPopupBehavior`: Enables a command component to launch a popup component. For details, see Section 13.4, "Invoking Popup Elements."
- `showPrintablePageBehavior`: Enables a command component to generate and display a printable version of the page. For details, see Section 33.2, "Displaying a Page for Print."
- `scrollComponentIntoViewBehavior`: Enables a command component to jump to a named component when clicked. For details, see Section 5.6.1, "How to Use the scrollComponentIntoViewBehavior Tag."

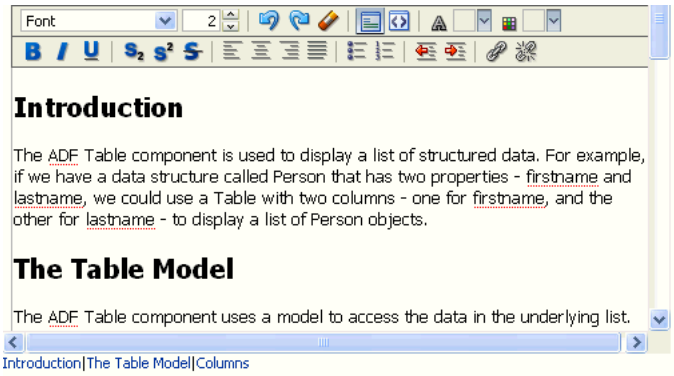
Client behavior tags cancel server-side event delivery automatically. Therefore, any `actionListener` or `action` attributes on the parent component will be ignored. This cannot be disabled. If you want to also trigger server-side functionality, you should use either a client-side event (see Section 5.3, "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 5.4, "Sending Custom Events from the Client to the Server").

5.6.1 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 5–1 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 5–1 scrollComponentIntoViewBehavior Tag in an Editor



You can also configure the tag to have focus switched to the component to which the user has scrolled.

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 18.2.1, "How to Use Command Buttons and Command Links."
2. In the Component Palette, from the **Operations** section, 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 Property Inspector, 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 5–1 is `Introduction`. The value of the `richTextEditor` is bound to the property shown in Example 5–17. Note that `Introduction` is the ID for the first header.

Example 5–17 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>";
```

5.7 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 a user 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.2.3.20, "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.

5.7.1 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 5.7, "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 2.6, "Creating and Using Managed Beans"
2. Create a poll component by dragging and dropping a **Poll** from the Operations panel of the Component Palette.
3. In the Property Inspector, 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.
4. If you want to override the global timeout value in the `web.xml` file, expand the Other section and set **Timeout** to the amount of time in milliseconds after which the page will stop polling and the session will time out.

6 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 handle and display any errors, including those not caused by validation.

This chapter includes the following sections:

6.1 Introduction to 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. If the `required` attribute of an input component is set to `true` you can set one or more validator attributes or you can use the ADF Faces validator components. 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 17, "Displaying Tips, Messages, and Help."

6.2 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 `sumbittedValue` attribute. If the value requires conversion (for example, if it is displayed as a `String` type but stored as a `DateTime` 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 4, "Using the JSF Lifecycle with ADF Faces."

6.3 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. Then the data is validated against any rules and conditions. 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 6–1 shows the converters provided by ADF Faces.

Table 6–1 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.
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 6–1, 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.

Some oracle.jbo.domain datatypes are automatically converted. For some oracle.jbo.domain datatypes that are not handled automatically, you can add a oracle.jbo.domain converter to your component as described in Section 6.3.4, "How to Add oracle.jbo.domain Converters."

6.3.1 How to Add a Standard ADF Faces Converter

You can also manually insert a converter into a UI component.

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.
2. In the context menu, choose **Insert inside <UI component>**, then **ADF Faces** to insert an ADF Faces converter, or **JSF Core** to insert a JSF converter.
3. Choose a converter tag (for example, **ConvertDateTime**).
4. In the Property Inspector, 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 6.3.2, "How to Set Attributes on a Standard ADF Faces Converter".

   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 17, "Displaying Tips, Messages, and Help."

6.3.2 How to Set Attributes on a Standard ADF Faces Converter

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. Not all converters support multiple patterns, although pattern matching is flexible and multiple patterns may not be needed.

Example 6–1 illustrates the use of a multiple pattern for the `af:convertColor` tag in which "255-255-000" and "FFFF00" are both acceptable values.

Example 6–1 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 6–2 illustrates the use of an `af:convertDateTime` tag in which "6/9/2007" and "2007/9/6" are both acceptable values.

Example 6–2 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 6–3 illustrates an `af:convertNumber` tag with the `type` attribute set to `currency` to accepts "$78.57" and "$078.57" as values for conversion.

Example 6–3 af:convertNumber Set to Currency Attribute

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

6.3.3 What Happens at Runtime

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 `value` 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.

6.3.4 How to Add oracle.jbo.domain Converters

For `oracle.jbo.domain` datatypes that are not automatically converted, you will need to reference the `oracle.jbo.domain` converter in your component. These converters are automatically registered and do not need a tag.

Table 6–2 lists the `oracle.jbo.domain` datatype converters.

Table 6–2 oracle.jbo.domain Datatype Converters

oracle.jbo.domain Converter	Description
`ordDomainConverter`	Handles `oracle.jbo.domain.ord` datatypes
`genericDomainConverter`	Handles generic `oracle.jbo.domain` datatypes

To add a `oracle.jbo.domain` converter, you can add the converter to the `converter` attribute as shown in Example 6–4.

Example 6–4 Adding genericDomain Converter using the converter attribute

```
<af:inputText  ...  converter="oracle.genericDomain"/>
```

Or you can add the `f:converter` tag and reference the converter using the `converterId` attribute as shown in Example 6–5.

Example 6–5 Adding genericDomain Converter using f:converter

```
<af:inputText ...
     <f:converter converterId="oracle.genericDomain"/>
</af:inputText>
```

6.4 Creating Custom JSF Converters

You can create your own converters to meet your specific business needs. You can create custom JSF converters that run on the server-side using Java, and then also create a JavaScript version that can run on the client-side. However, unlike creating custom validators, you can create only converter classes. You cannot add a method to a backing bean to provide conversion.

6.4.1 How to Create a Custom JSF Converter

Creating a custom converter requires writing the business logic for the conversion by creating an implementation of the `Converter` interface that contains the `getAsObject()` and `getAsString()` methods, and then registering the custom

converter with the application. You then use the `f:converter` tag and set the custom converter as a property of that tag, or you can use the `converter` attribute on the input component to bind to that converter.

You can also create a client-side version of the converter. ADF Faces client-side converters work in the same way standard JSF conversion works on the server, except that JavaScript is used on the client. JavaScript converter objects can throw `ConverterException` exceptions and they support the `getAsObject()` and `getAsString()` methods.

To create a custom JSF converter:

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 API documentation 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 API documentation for `javax.faces.convert.ConverterException` 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 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.

4. Register the converter in the `faces-config.xml` file.

   - Open the `faces-config.xml` file and select the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory in the JDeveloper Application Navigator.

- In the window, select **Converters** and click **New**. Click **Help** or press **F1** for additional help in registering the converter.

To create a client-side version of the converter:

- Write a JavaScript version of the converter, passing relevant information to a constructor. Example 6–6 shows the code to implement the interface `org.apache.myfaces.trinidad.convert.ClientConverter`, which has two methods. The first method is `getClientScript()`, which returns an implementation of the JavaScript `Converter` object. The second method is `getClientConversion()`, which returns a JavaScript constructor that is used to instantiate an instance of the converter.

Example 6–6 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){}
```

The `TrConverter` interface can throw a `TrConverterException` exception, which should contain a `TrFacesMessage` error message. Example 6–7 shows the signature for `TrFacesMessage` and Example 6–8 shows the signature for `TrFacesException`.

Example 6–7 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 6–8 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
```

Example 6–9 shows an example of a customer converter, SimpleNumberConverter, written in Java that will run on the server. The custom converter has to implement the `ClientConverter` interface.

Example 6–9 Custom Converter SimpleNumberConverter in Java

```
public class SimpleNumberConverter implements javax.faces.convert.Converter,
 org.apache.myfaces.trinidad.convert.ClientConverter
{
 public SimpleNumberConverter(boolean isInteger)
 {
    _isInteger = isInteger;
 }

 // CONVERTER INTERFACE
 public Object getAsObject(FacesContext context, UIComponent component,
                                              String value)
 {
    // convert to object
 }

 String getAsString(FacesContext context, UIComponent component, Object value)
 {
   // convert to string
 }
 // CLIENTCONVERTER INTERFACE
 /**
  * Called to retrieve the appropriate client
  * conversion code for the node and context.
  * For HTML, this will be javascript that will be embedded in a
  * script tag. For HTML this method is expected to return a
  * constructor of the javascript Converter object
  * returned by getClientScript().
  */
 public String getClientConversion(FacesContext context, UIComponent component)
 {
    return "new SimpleNumberConverter(" + Boolean.toString(_isInteger) + ")";
 }
 public Collection<String> getClientImportNames()
 {
   // return empty collection
 }
 public String getClientLibrarySource(FacesContext context)
```

```
 {
   return null;
 }

 public String getClientScript(FacesContext context, UIComponent component)
 {
   return null;
 }

 private boolean _isInteger;
}
```

You must also create a JavaScript implementation of the custom converter for the client, as shown in Example 6–10.

Example 6–10 Client-side Implementation of SimpleNumberConverter in JavaScript

```
/**
* constructor of client side SimpleNumberConverter class
*/
function SimpleNumberConverter(isInteger)
{
 this._isInteger = isInteger;
}

// Inherit object properties from base class if desired.
SimpleNumberConverter.prototype = new SimpleConverter();

SimpleNumberConverter.prototype.getAsString = function(number,label)
{
 // convert to string
}

SimpleNumberConverter.prototype.getAsObject = function(numberString,label)
{
 // convert to object
}
```

To use a custom converter on a JSF page:

- Bind your converter class to the `converter` attribute of the input component.

> **Note:** If a custom 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}"/>
> ```
>
> The `myProperty` data type has the same type as the custom converter.

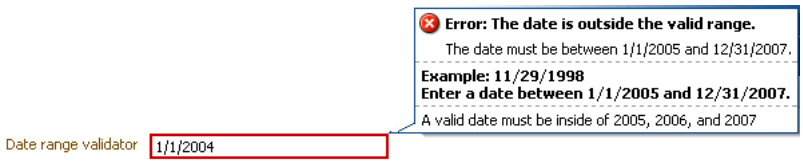
6.4.2 What Happens When You Use a Custom Converter

When you use a custom converter, the application accesses the converter class referenced in the `converter` attribute, and executes the `getAsObject` or `getAsString` method as appropriate. These methods access the data from the component and execute the conversion logic.

6.5 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 6–1 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 6–1 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 6.5.1.2, "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 6.5.1.3, "Using ADF Faces Validators." You can also create your own validators. For information about custom validators, see Section 6.6.3, "How to Create a Custom JSF Validator."

6.5.1 How to Add Validation

Set ADF Faces validation on the input component and an error message is displayed inline or in a popup window on the page. For more information about displaying messages created by validation errors, see Chapter 17, "Displaying Tips, Messages, and Help."

6.5.1.1 Adding ADF Faces Validation

By default, ADF Faces syntactic and semantic validation occurs on both the client and server side. Client-side validation allows validators to catch and display data without requiring a round-trip to the server.

ADF Faces provides the following types of validation:

- UI component attributes: ADF Faces input components provide attributes that can be used to validate data. For example, you can supply simple validation using the `required` attribute on ADF Faces input components to specify whether or not a value must be supplied. When the `required` attribute is set to `true`, the component must have a value. Otherwise the application displays an error message. For more information, see Section 6.5.1.2, "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 6.5.1.3, "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 6.6, "Creating Custom JSF Validation."

6.5.1.2 Using Validation Attributes

Many ADF Faces UI components have attributes that provide simple validation. For example, the `af:chooseDate` component is used in conjunction with an `af:inputDate` component for easy date selection. The `af:chooseDate` 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 Property Inspector, right-click the attribute name and choose **Help**.

6.5.1.3 Using ADF Faces Validators

ADF Faces Validators are separate classes that can be run on the server or client. Table 6–3 describes the validators and their logic.

Table 6–3 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.

Table 6–3 (Cont.) ADF Faces Validators

Validator	Tag Name	Description
`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 6.6, "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.
2. In the context menu, choose **Insert inside <UI component>**, then **ADF Faces** to insert an ADF Faces validator, or **JSF Core** to insert a JSF reference implementation validator.
3. Choose a validator tag (for example, **ValidateDateTimeRange**).
4. In the Property Inspector, 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.

6.5.2 What Happens at Runtime

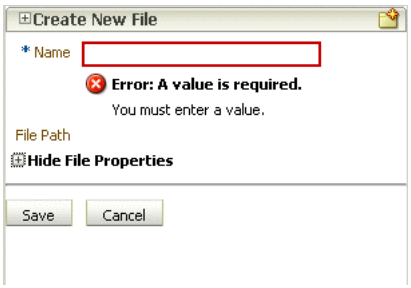
When the user submits the page, ADF Faces checks the submitted value and runs conversion on any non-null value. The converted value is then passed to the `validate()` method. If the value is empty, the `required` attribute of the component is checked and an error message is generated if indicated. If 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. 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. ADF Faces automatically displays an error icon next to the label of any input component that generated an error, and displays the associated messages in a popup window unless the `af:message` component `inline` attribute is set to `true`. Figure 6–2 shows a server-side validation error.

Figure 6–2 Server-Side Validation Error



6.5.3 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 4, "Using the JSF Lifecycle with ADF Faces."

6.6 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.

6.6.1 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.

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 to open the Bind Validator Property dialog.
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 API documentation for `javax.faces.validator.ValidatorException` and `javax.faces.application.FacesMessage`, or visit `http://docs.oracle.com/javaee/index.html`.

6.6.2 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 6–11 shows the code JDeveloper generates.

Example 6–11 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.

6.6.3 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.

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 API documentation 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 API documentation 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.
   - Open the faces-config.xml file and select the **Overview** tab in the editor window. The faces-config.xml file is located in the <View_Project>/WEB-INF directory.
   - In the window, select **Validators** and click **New**. 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 6–12 shows a validator in Java.

Example 6–12 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 6–13.

Example 6–13 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 6–14 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 6–14 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*** > **ADF Faces Core** > **Validator**.
2. 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 6–15 shows the code on a JSF page for a validator using the `f:validator` tag.

Example 6–15 A Custom Validator Nested Within a Component on a JSF Page

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

6.6.4 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 accesses the data from the component in the current `FacesContext` and executes logic against it 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.

7

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:

7.1 Introduction to 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). PPR allows certain components on a page to be rerendered without the need to rerender the entire page. For example, an output component can display what a user has chosen or entered in an input component, or a command link or button can cause another component on the page to be rerendered, without the whole page rerendering.

In order for PPR to work, boundaries must be set on the page that allow the lifecycle to run just on components within the boundary. In order to determine the boundary, the framework must be notified of the root component to process. The root component can be identified 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 8.8, "Displaying and Hiding Contents Dynamically"), indicates that the `showDetail` component is a root. When the `showDetail` component is expanded or collapsed, only that component 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.
- Components: Certain components are recognized as a 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.

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. When an event occurs on the trigger component, the lifecycle is run only on listener components and child components to the listener, and only the listener components and their children are rerendered. Cross-component rendering can be implemented declaratively. However, by default, all events from a trigger component will cause PPR (note that some components, such as table, trigger partial targets on only a subset of their events). For these cases where you need strict control over the event that launches PPR, or for cases where you want to use some logic to determine the target, you can implement PPR programatically.

> **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 Automatic Partial Page Rendering" section of the *Oracle Fusion Middleware Fusion Developer's Guide for 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.

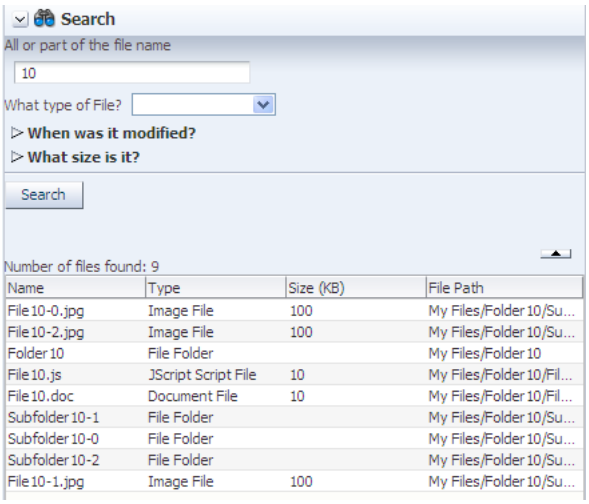
Note: The browser must have JavaScript enabled for PPR to work.

7.2 Enabling Partial Page Rendering Declaratively

Using the simplest form of cross-component rendering, 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 7–1, the File Explorer application contains a table that shows the search results in the Search panel. This table (and only this table) is rerendered when the search button is activated. The search button is configured to be the trigger and the table is configured to be the target.

Figure 7–1 The Search Button Causes Results Table to Rerender



Note: In some cases, you may want a component to be rerendered only when a particular event is fired, not for every event associated with the trigger component, or you may want some logic to determine whether a component is to be rerendered. In these cases, you can programatically enable PPR. For more information, see Section 7.3, "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 `commandButton` component, and an `outputText` component. When the user enters a value for the `inputText` component, and then clicks the `commandButton` component, the input value is reflected in the `outputText` component. You would set the `partialSubmit` attribute to `true` on the `commandButton` 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 `commandButton` 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. However, 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 4.3, "Using the Optimized Lifecycle."

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 3.5, "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: 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.

Note: If your trigger component is an `inputLov` or an `inputComboBoxLov`, and the target component is an input component set to `required`, then a validation error will be thrown for the input component when the LOV popup is displayed. To avoid this, you must use programmatic partial page rendering. For more information, see Section 7.3, "Enabling Partial Page Rendering Programmatically."

7.2.1 How to Enable Partial Page Rendering

For a component to be rerendered based on an event caused by another component, it must declare which other components are the triggers.

To enable a component to rerender another component:

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 Property Inspector 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 3.5, "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 Property Inspector, 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 Property Inspector, 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 Property Inspector, click the dropdown menu for 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 7–1 shows a `commandLink` component configured to execute PPR.

Example 7–1 Code for Enabling Partial Page Rendering Through a Partial Submit

```
<af:commandLink id="deleteFromCart" partialSubmit="true"
                actionListener="#{homeBean...}">
```

Example 7–2 shows an `outputText` component that will be rerendered when the command link with ID `deleteFromCart` in Example 7–1 is clicked.

Example 7–2 Code for Partial Page Rendering Triggered by Another Component

```
<af:outputText id="estimatedTotalInPopup"
               partialTriggers="deleteFromCart"
               value="#{shoppingCartBean...}"/>
```

Tip: You can use PPR to prevent components from being validated on a page. For more information, see Section 4.3, "Using the Optimized Lifecycle."

7.2.2 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.

7.2.3 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.

7.3 Enabling Partial Page Rendering Programmatically

For components such as calendars that have many associated events, PPR will happen any time any event is triggered, causing any component with the calendar as a partial trigger to be rerendered with each event. If you want the target to be rerendered only for certain events, or if you want a target to be rerendered based on some other logic, you can enable partial page rendering programmatically.

For example, in the ADF Faces calendar demo, if a user attempts to change the duration of an activity that no longer exists in the model, the calendar needs to be refreshed to display without the activity (the calendar automatically refreshes itself if a valid activity's duration is changed). In this example, the `activityDurationChangeListener` method sets the calendar as a partial target whenever the `activityDurationChangeEvent` is invoked, and the `activity` object is null.

Before you begin:

Create a managed bean that will contain the listener method. For more information, see Section 2.6, "Creating and Using Managed Beans."

How to enable PPR programatically:

1. In the JSF page, select the target component. In the Property Inspector, enter the set **ClientComponent** to **true**.

   Note: You must set the `clientComponent` attribute to `true` to ensure that the client ID will be generated.

2. Use the `binding` attribute so that the managed bean can work with an instance of the target component. To do so:

   1. In the Property Inspector, set the `Binding` to an EL expression that resolves to the target component on the managed bean.

      In the above example, you might set `Binding` to:

      ```
      #{myBean.cal1}
      ```

      Where `cal1` is the ID of the target component, in this case, the `calendar` component.

   2. In the managed bean, create get and set methods for the target component. Example 7–3 shows what the code on a managed bean might look like for the `calendar` component.

Example 7–3 Get and Set Methods for a UI Component

```
public class MyBean {
    private RichCalendar cal1;

    public MyBean() {
    }

  public void setCal1(RichCalendar cal1) {
      this.cal1 = cal1;
  }

  public RichOutputText getCal1() {
      return cal1;
  }
```

3. In the managed bean, create a listener method for the event on the trigger component 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.

   Example 7–4 shows how you might create a `ActivityDurationChangeEvent` listener that adds the calendar as a target.

Example 7–4 Rerendering Using Partial Targets

```
public void activityDurationChangeListener(CalendarActivityDurationChangeEvent ae)
  {
    CalendarActivity activity = ae.getCalendarActivity();

    if (activity == null)
    {
      // no activity with that id is found in the model
```

```
      System.out.println("No activity with event " + ae.toString());
      setCurrActivity(null);

      // Since the user has acted on an activity that couldn't be found,
      // ppr the page so that they no longer see the activity
      RequestContext adfContext = RequestContext.getCurrentInstance();
      adfContext.addPartialTarget(getCal1());
      return;
    }

    DemoCalendarActivity demoActivity = ((DemoCalendarActivity)activity);
    TimeZone tz = getTimeZone();
    demoActivity.setEndDate(ae.getNewEndDate(), tz);
    setCurrActivity(new DemoCalendarActivityBean(demoActivity, tz));
  }
```

4. Select the trigger component, and in the Property Inspector, find the listener for the event that will cause the refresh and bind it to the listener method created in Step 3.

7.4 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 partial page rendering request. The new page is sent to the client using partial page rendering. Partial page navigation is disabled by default.

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.

7.4.1 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`.

To use partial page navigation:

1. Double-click the `web.xml` file.
2. In the source editor, change the `oracle.adf.view.rich.prNavigation.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.

7.4.2 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.
- Because PPR navigation makes use of the hash portion of the URL, you cannot use the hash portion for navigation to anchors within the 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

Using ADF Faces Components

Part III contains the following chapters:

8

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:

8.1 Introduction to 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 19, "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 8.2.1, "Geometry Management and Component Stretching."

Table 8–1 briefly describes each of the ADF Faces layout components.

Table 8–1 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 8.2, "Starting to Lay Out a Page."	X	
`form`	Creates an HTML `<form>` element. For more information, see Section 8.2, "Starting to Lay Out a Page."		
Page Layout Containers			
`panelStretchLayout`	Contains `top`, `bottom`, `start`, `center`, and `end` facets where you can place other components. For more information, see Section 8.3, "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 8.4, "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 8.7, "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 8.5, "Arranging Page Contents in Predefined Fixed Areas."		

Table 8–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
`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 8.6, "Arranging Content in Forms."		
Components with Show/Hide Capabilities			
`showDetailHeader`	Can hide or display contents below the header. Often used as a child to the `panelHeader` component. For more information, see Section 8.8, "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 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed Panels."	X (if it contains a single child component)	
`panelBox`	Titled box that can contain child components. Has a toolbar facet. For more information, see Section 8.8, "Displaying and Hiding Contents Dynamically."		X
`panelAccordion`	Used in conjunction with `showDetailItem` components to display as a panel that can be expanded or collapsed. For more information, see Section 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed 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 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed 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 18.5, "Using Navigation Items for a Page Hierarchy."		X (when the `dimensionsFrom` attribute is set to `parent`)

Table 8–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
`showDetail`	Hides or displays content through a toggle icon. For more information, see Section 8.8, "Displaying and Hiding Contents Dynamically."		
Miscellaneous Containers			
`panelHeader`	Contains child components and provides a header that can include messages, toolbars, and help topics. For more information, see Section 8.10, "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 10.8, "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 typically acts as a look and feel transition between areas on a page. For example, a page that has a dark background for its template can use the decorative box to transition to a white background for its main area. For more information, see Section 8.10, "Displaying Items in a Static Box."	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 18.5, "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 8.11, "Displaying a Bulleted List in One or More Columns."		
`panelWindow`	Displays child components inside a popup window. For more information, see Section 13.2, "Declaratively Creating Popup Elements."		

Table 8–1 (Cont.) ADF Faces Layout Components

Component	Description	Can Stretch Children	Can Be Stretched
`toolbox`	Displays child toolbar and menu components together. For more information, see Section 14.3, "Using Toolbars."		
Grouping Containers			
`panelGroupLayout`	Groups child components either vertically or horizontally. Used in facets when more than one component is to be contained in a facet. For more information, see Section 8.12, "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. Used in facets when more than one component is to be contained in a facet. For more information, see Section 8.12, "Grouping Related Items."		
Spacing Components			
`separator`	Creates a horizontal line between items. For more information, see Section 8.13, "Separating Content Using Blank Space or Lines."		
`spacer`	Creates an area of blank space. For more information, see Section 8.13, "Separating Content Using Blank Space or Lines."		

8.2 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 8.2.5, "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 8.6, "Arranging Content in Forms."

JDeveloper automatically inserts the `view`, `document`, and `form` tags for you, as shown in Example 8–1. For more information, see Section 2.4, "Creating a View Page."

Example 8–1 Initial JSF Page Created by JDeveloper Wizard

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          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">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
    <af:document>
      <af:form/>
    </af:document>
  </f:view>
</jsp:root>
```

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 8.2.1, "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 8.2.3, "Using Quick Start Layouts."

8.2.1 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. At the heart of the RCF geometry management solution is a resize notification mechanism that allows components that support geometry management to be notified of browser resize activity. The following scenarios trigger the notification:

- Load: When the page contents are first loaded, allowing initial sizing to take place.
- Browser resize: When the browser window is resized.
- Partial replacement: When a portion of the page is updated through partial page rendering, any newly inserted components are notified, allowing them to perform any necessary geometry management.
- Visibility change: When a component that was initially configured to be invisible is made visible (components that are initially not visible do not receive notification).

- Explicit resize: When components that allow themselves to be resized (for example the `panelSplitter`), are resized by the user.

By default, the root component will stretch automatically to consume the browser's viewable area, provided that component supports geometry management and therefore can stretch its child components. Examples of geometry management components are `panelStretchLayout` and `panelSplitter`.

> **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 13, "Using Popup Dialogs, Menus, and Windows."

When the user resizes the browser window, and when there is a single maximized root visual component inside of the `document` component, that visual root component will also resize along with the browser window. 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.

As shown in Table 8–1, the `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 8–1 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 8–1 Table Inside a Component That Stretches Child Components

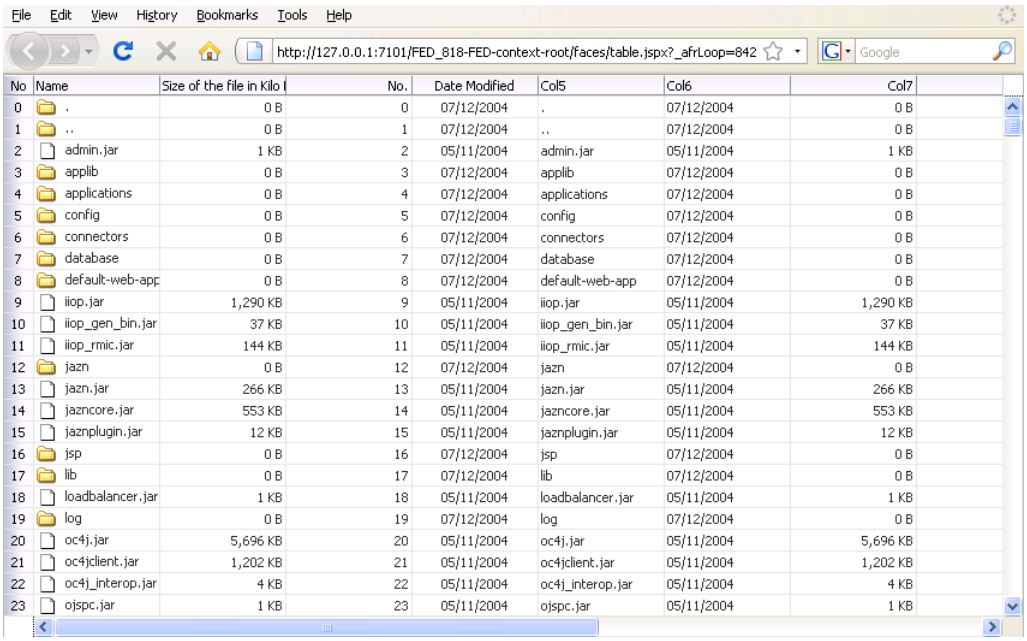
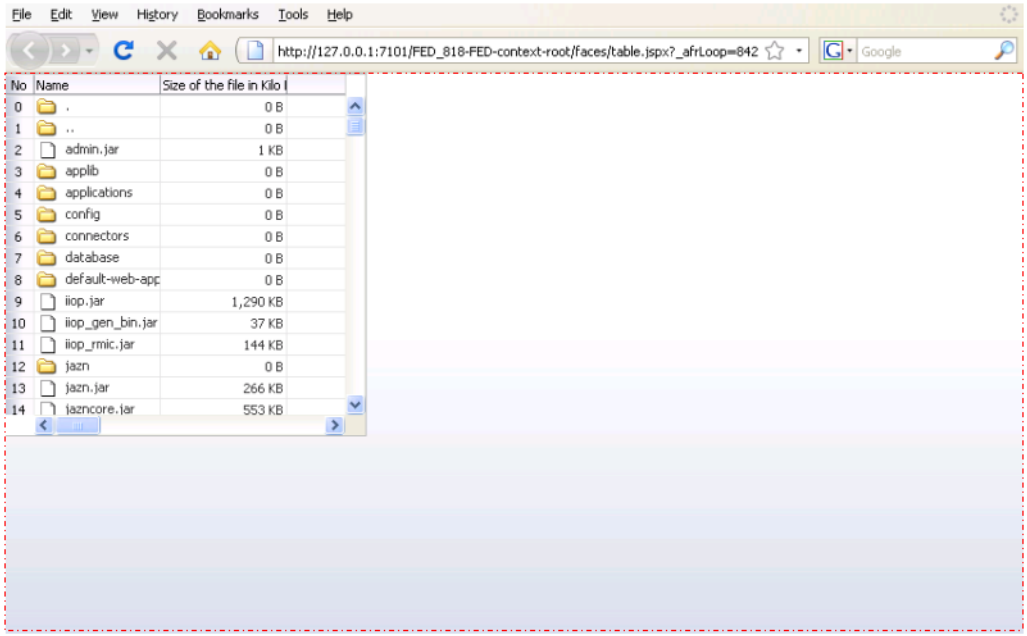


Figure 8–2 shows the same table, but nested inside a `panelGroupLayout` component, which cannot stretch its child components (for clarity, a dotted red outline has been placed around the `panelGroupLayout` component). The table component displays only a certain number of columns and rows, determined by properties on the table.

Figure 8–2 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.

8.2.2 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:

- `inputText` (when configured to stretch)
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection`
- `panelDashboard` (when configured to stretch)
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (with the `layout` attribute set to `scroll` or `vertical`)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following 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`)
- `panelHeader`
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `showDetailHeader`
- `tableLayout` (MyFaces Trinidad component)

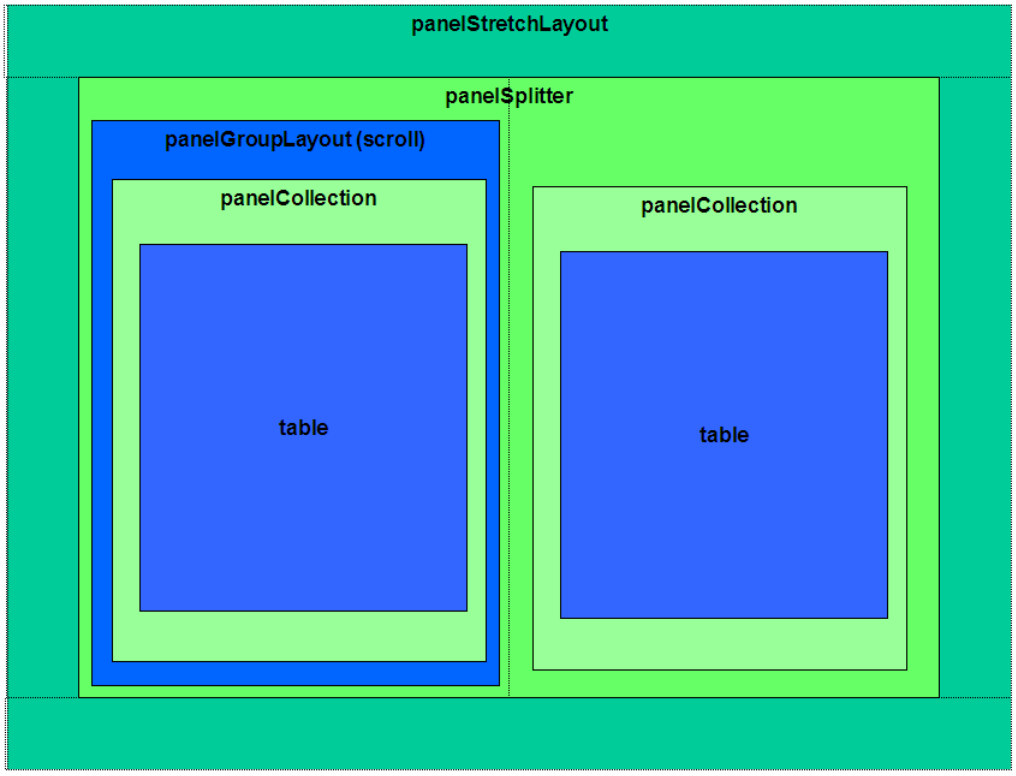
Because these components cannot be stretched, you cannot place them in a facet of any component that stretches its child components. So if you want to use one of these components within the facet of component that does stretch its child components, you must wrap it in a component that can be stretched, but one that does not stretch its child components. If you do not, you may see unexpected results when the component renders.

For example, suppose you have a `panelStretchLayout` as the first component on your page. You then add a `panelSplitter` component that is configured to be stretched. Now to the first facet of the `panelSplitter` component, say you add a `panelGroupLayout` component with its `layout` attribute set to `scroll` (which

means it can stretch), and inside that, you add a `panelCollection` component, and then finally a `table` component.

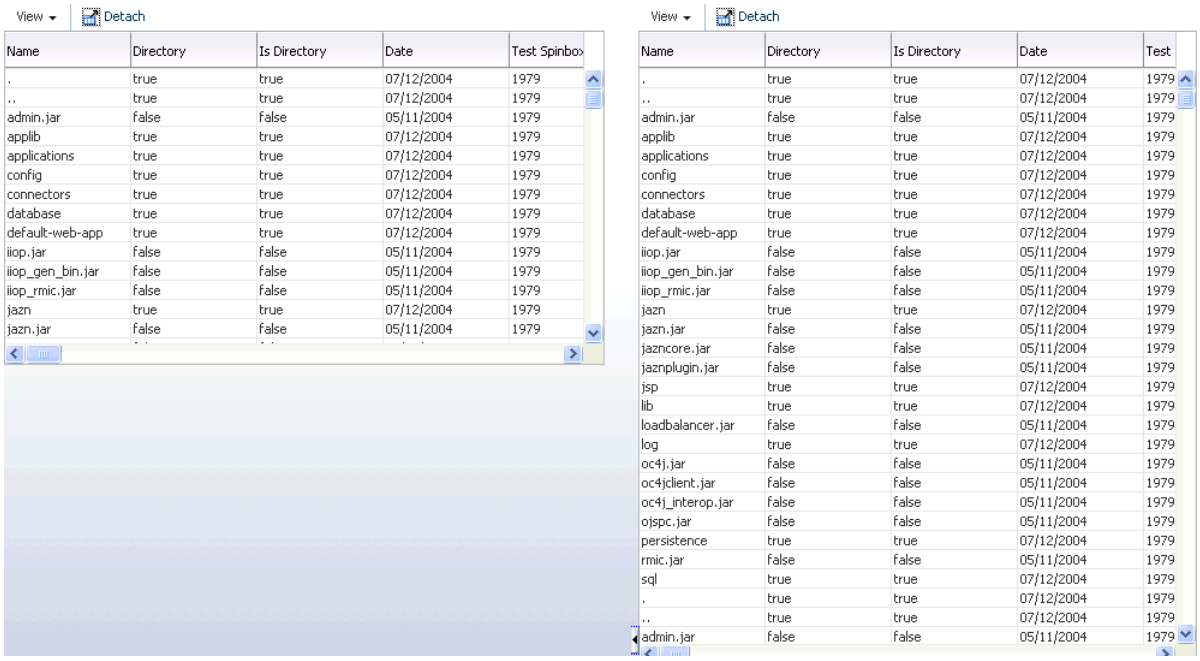
To the second facet of the `panelSplitter`, suppose you add just the `panelCollection` and `table` components, as shown in Figure 8–3. Components that can stretch their children are green and components that can be stretched (but cannot stretch their children) are blue.

Figure 8–3 Layout Using Geometry-Managed Components



As shown in Figure 8–4, when the page is run, the `panelCollection` and `table` components in the `panelGroupLayout` do not stretch, while the ones directly in the `panelSplitter` component do stretch.

Figure 8–4 Geometry-Managed Components Affect the Layout of the Page



Because the `panelStretchLayout` component can stretch its child components, and because the `panelSplitter` component was configured to stretch, both stretch to fill up available browser space. Because `panelSplitter` component can stretch its child components and because on the left, `panelGroupLayout` component with its `layout` attribute set to `scroll` can be stretched, and on the right, the `panelCollection` component can be stretched, both of those stretch to fill up available browser space. However, the `panelGroupLayout` component cannot stretch its child components, while the `panelCollection` component can stretch a single table. Therefore, the `panelCollection` component on the left does not stretch, even though its parent does.

> **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. For components that can be stretched, see Table 8–1.

Now suppose on the left, instead of a `table` component, you want to add a `panelList` component. You would not need the `panelCollection` component (as that is used only for tables), so you might also think you would not need to use the `panelGroupLayout` component to group the `panelList` component with another component. However, because the `panelList` component would then be a direct child of the `panelSplitter` component, and because the `panelSplitter` component stretches its child components and the `panelList` component cannot be stretched, you would need to keep the `panelGroupLayout` (set to `scroll`) and place the `panelList` component as a child to the `panelGroupLayout` component.

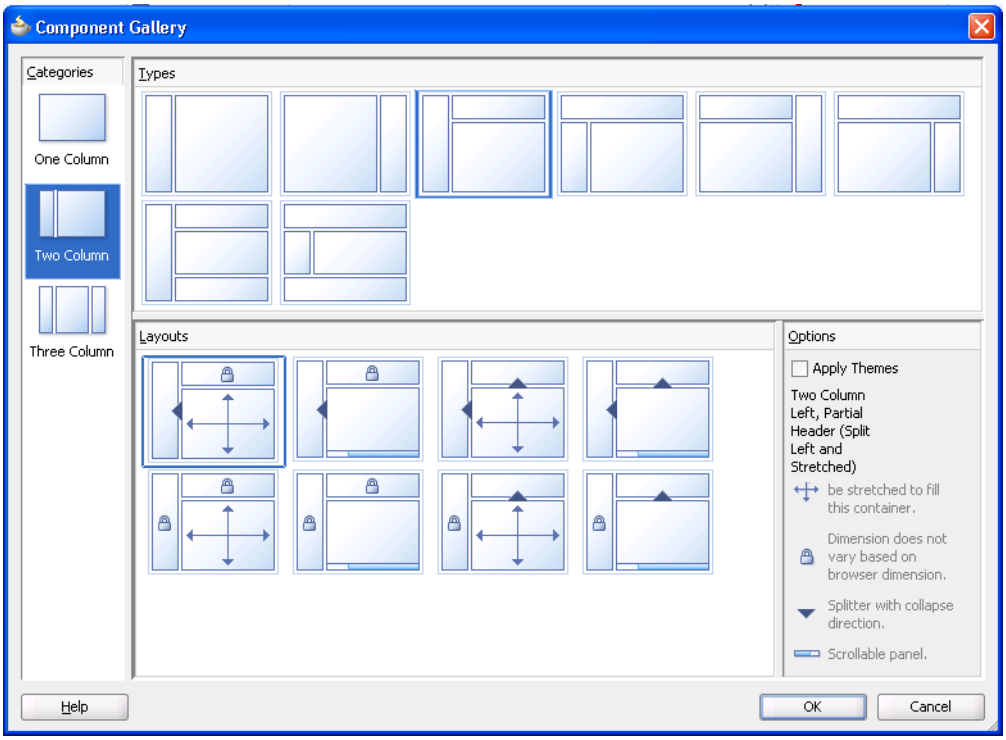
This way, the `panelSplitter` component can stretch the `panelGroupLayout` component, but the `panelGroupLayout` component will not try to stretch the `panelList` component. Because the `panelGroupLayout` component can be stretched, but does not stretch its child components, it allows the transition between a layout that stretches and one that flows.

Components that can be stretched but do not stretch their children are considered *transition* components. Transition components must always be used between a component that stretches its children and a component that does not stretch.

8.2.3 Using Quick Start Layouts

When you use the New Gallery Wizard to create a JSF JSP 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. 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 8–5 shows the different layouts available in the two-column format.

Figure 8–5 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 Section 20.3.4, "How to Apply Themes to Components."

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 outcome. For more information about creating pages using the quick layouts, see Section 2.4, "Creating a View Page."

8.2.4 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 ensure that the correct

components are used and configured properly. For more information, see Section 8.2.3, "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.
- 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 8.2.5, "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.

8.2.5 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 tag so that one component has focus when the page is first rendered. You can also configure the tag to display a warning message if a user attempts to navigate off the page and the data has not been submitted. You can also set the document to use a different state saving method than the rest of the application.

To configure the document tag:

1. In the Structure window, select the **af:document** node.
2. In the Property Inspector, 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 3.4, "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 8.2.1, "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 for the **FailedConnectionText** attribute, enter the text you want to be displayed if a connection cannot be made to the server.
4. Expand the Other section and set the following:
   - **UncommittedDataWarning**: Set 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`.
   - **StateSaving**: Set to the type of state saving you want to use for a page.

     For ADF Faces applications, it is recommended to have the application 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.

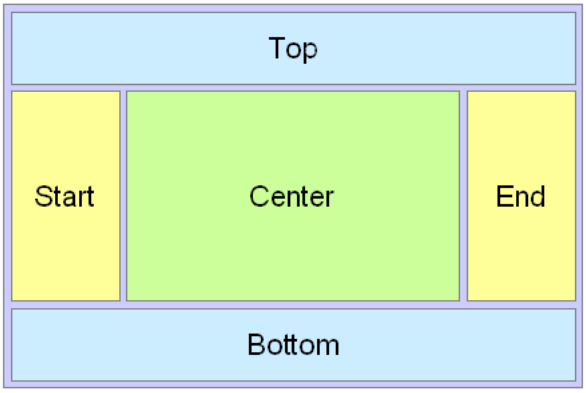
     You can override the global setting in `web.xml` to one of the following for the page:
     - **client**: The state is saved fully to the client, without the use of tokens. This setting keeps the session expired messages from being displayed.
     - **default**: The state of the page is based on whatever 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."

8.3 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 8–6 shows the component's facets.

Figure 8–6 Facets in the panelStretchLayout Component



Note: Figure 8–6 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 8.13, "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 8.2.1, "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 8–2 shows abbreviated code from the `fileExplorerTemplate` file.

Example 8–2 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.

8.3.1 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 8.2.2, "Nesting Components Inside Components That Allow Stretching."

To create and use the panelStretchLayout component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Stretch Layout** to the JSF page.
2. In the Property Inspector, 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. By default, the `panelStretchLayout` component stretches to fill available browser space. If you want to place the `panelStretchLayout` component inside a component that does *not* stretch its children, then you need to configure the `panelStretchLayout` component to not stretch.

   You configure whether the component will stretch or not using the `dimensionsFrom` attribute. To do so, expand the **Other** section, and 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 8.3.2, "What You May Need to Know About Geometry Management and the panelStretchLayout Component," for more details.

   Because facets accept one child only, if you want to add more than one child component, wrap the child components inside a container component, for example, a `panelGroupLayout` component. This component 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.

8.3.2 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:

- `inputText` (when configured to stretch)
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `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`)
- `panelSplitter` (when configured to stretch)

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

The following components cannot be stretched when placed inside a facet of the `panelStretchLayout` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelHeader`
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `showDetailHeader`
- `tableLayout` (MyFaces Trinidad component)

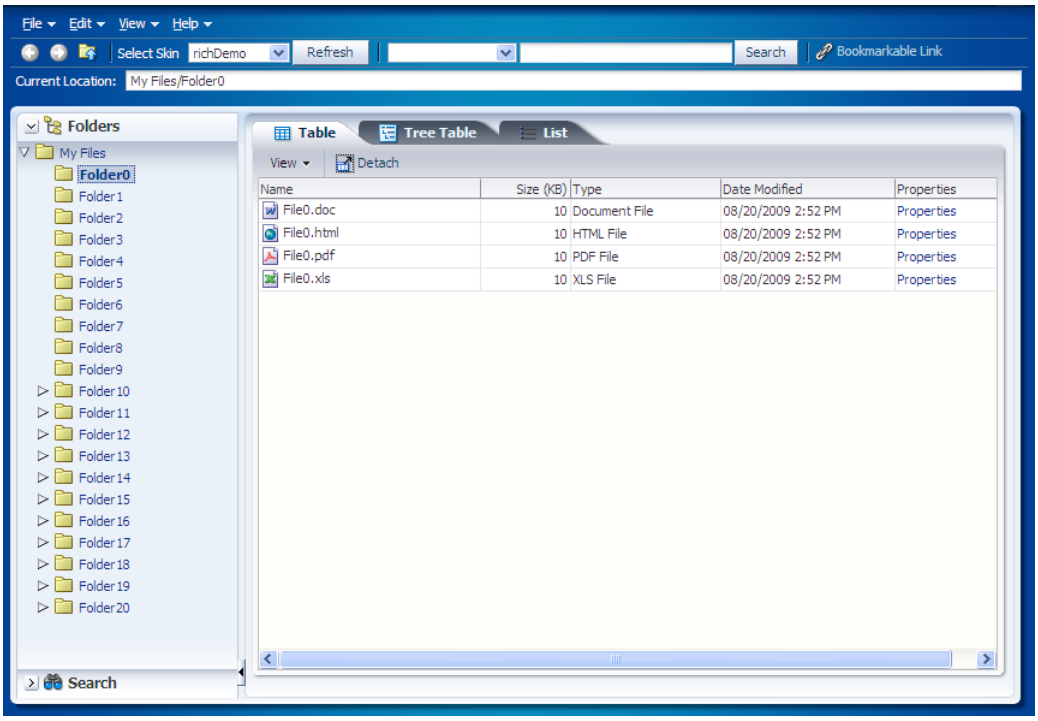
You cannot place components that cannot stretch into facets of a component that stretches its child components. Therefore, if you need to place a component that cannot be stretched into a facet of the `panelStretchLayout` component, wrap that component in a transition component that can stretch.

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 8.2.2, "Nesting Components Inside Components That Allow Stretching."

8.4 Using Splitters to Create Resizable Panes

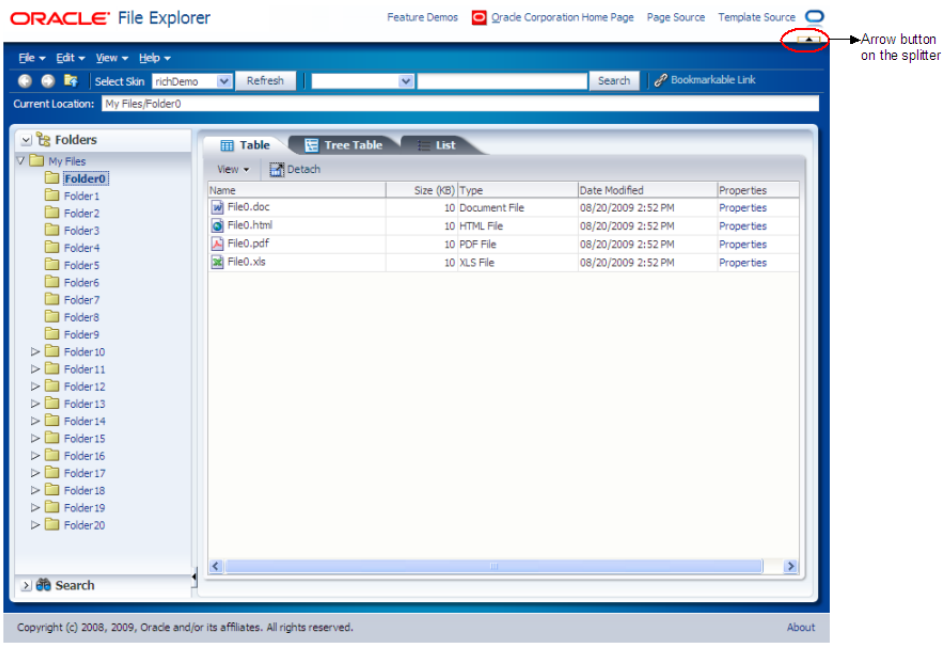
When you have groups of unique content to present to users, consider using the `panelSplitter` component to provide multiple panes separated by adjustable splitters. The File Explorer uses a `panelSplitter` to separate the navigation tree from the folder contents, as shown in Figure 8–7. Users can change the size of the panes by dragging the splitter, and can also collapse and restore the panel that displays the directories. When a panel is collapsed, the panel contents are hidden; when a panel is restored, the contents are displayed.

Figure 8–7 File Explorer Uses panelSplitter to Separate Contents



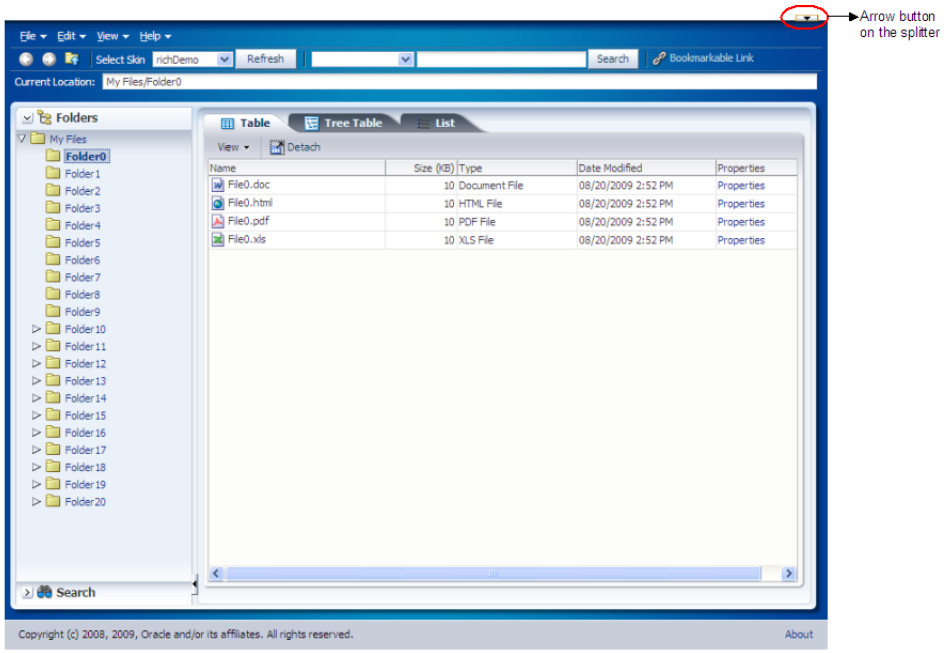
The `panelSplitter` component lets you organize contents into two panes separated by an adjustable splitter. The panes can either line up on a horizontal line (as does the splitter shown in Figure 8–7) or on a vertical line. The File Explorer application uses another `panelSplitter` component to separate the application's header contents from the main body of the page. Figure 8–8 shows the `panelSplitter` component expanded to show the header contents, which includes the Oracle logo and the File Explorer name.

Figure 8–8 panelSplitter with a Vertical Split Expanded



Clicking the arrow button on a splitter collapses the panel that holds the header contents, and the logo and name are no longer shown, as shown in Figure 8–9.

Figure 8–9 File Explorer Uses panelSplitter with a Vertical Split



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 31, "Allowing User Customization on JSF Pages."

8.4.1 How to Use the panelSplitter Component

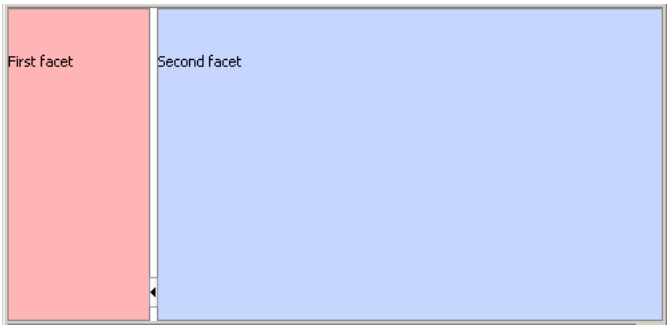
The `panelSplitter` component lets you create two panes separated by a splitter. Each splitter component has two facets, namely, `first` and `second`, which correspond to the first panel and second panel, respectively. Child components can reside inside the facets only. To create more than two panes, you nest the `panelSplitter` components.

To create and use the panelSplitter component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Splitter** onto the JSF page.
2. In the Property Inspector, expand the Common section.

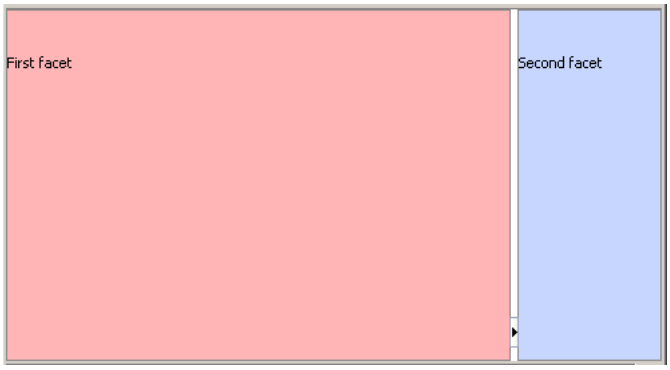
3. Set **Orientation** to `vertical` to create two vertical panes (one on top of the other). By default, the value is `horizontal`, which means horizontal panes are placed left-to-right (or right-to-left, depending on the language reading direction).
4. Set **SplitterPosition** and **PositionedFromEnd** 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 8–10.

Figure 8–10 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 8–11 shows the position of the splitter measured 200 pixels from the end panel.

Figure 8–11 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. To do so, expand the **Other** section, and 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 8.4.2, "What You May Need to Know About Geometry Management and the panelSplitter Component."

   Because facets accept one child component only, if you want to add more than one child component, wrap the child components inside a container component. This component 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 `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 8–12).

Figure 8–12 Nested panelSplitter Components

Example 8–3 shows the code generated by JDeveloper when you nest splitter components.

Example 8–3 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 5, "Handling Events."

8.4.2 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:

- `inputText` (when configured to stretch)
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection`
- `panelDashboard`
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a facet of the `panelSplitter` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only with the `layout` attribute set to `default` or `horizontal`)
- `panelHeader`
- `panelLabelAndMessage`
- `panelList`
- `showDetail`
- `showDetailHeader`
- `tableLayout` (MyFaces Trinidad component)

You cannot place components that cannot stretch 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

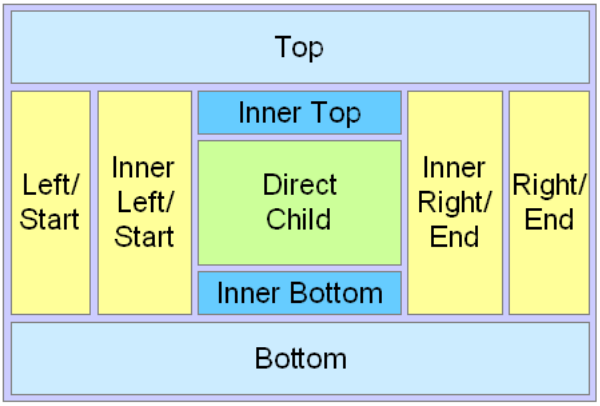
the panelSplitter component, and then place the panelBox component in that panelGroupLayout component. For more information, see Section 8.2.2, "Nesting Components Inside Components That Allow Stretching."

8.5 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 8–13 shows the facets of the panelBorderLayout component.

Figure 8–13 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 8.3, "Arranging Contents to Stretch Across a Page."

8.5.1 How to Use the panelBorderLayout Component

There is no restriction to the number of panelBorderLayout components you can have on a JSF page.

To create and use the panelBorderLayout component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Border Layout** onto the JSF page.
2. From the Component Palette, 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 8.12, "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 accept one child component only, if you want to add more than one child component, wrap the child components inside a container.

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.

8.6 Arranging Content in Forms

The `panelFormLayout` component lets you lay out multiple form input 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 8–14.

Figure 8–14 Right-Aligned Labels and Left-Aligned Fields in a Form

Figure 8–15 shows the same page with the component configured to display the labels above the fields.

Figure 8–15 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. For example, if there are 25 child components, and you set the component to display 15 rows, the last 10 components will be displayed in a second column.

However, the number of rows displayed in each is not solely determined by the configured number of rows. By default, the `panelFormLayout` component 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, it 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.

8.6.1 How to Use the panelFormLayout Component

You can use one or more `panelFormLayout` components on a page to create the desired form layout.

To create and use panelFormLayout:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Form Layout** onto the JSF page.
2. In the Property Inspector, 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 form.

   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 form. 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 form 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 8–4 shows the `panelFormLayout` component as it is used on the `properties.jspx` page of the File Explorer application, shown in Figure 8–14.

Example 8–4 panelFormLayout Component

```
<af:panelFormLayout rows="5" labelAlignment="top">
  <af:inputText value="#{fileItemProperties.type}"
                label="#{explorerBundle['fileproperties.type']}"
                readOnly="true"/>
  <af:inputText value="#{fileItemProperties.location}"
                label="#{explorerBundle['fileproperties.currentpath']}"
                readOnly="true"/>
  <af:inputText value="#{fileItemProperties.size}"
                label="#{explorerBundle['fileproperties.size']}"
                readOnly="true"/>
  <af:inputText value="#{fileItemProperties.contains}"
                label="#{explorerBundle['fileproperties.contains']}"
                readOnly="true"/>
```

```
</af:panelFormLayout>
```

> **Tip:** If you use non-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 17.4, "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 cause the `panelFormLayout` component to draw a separator line above and below the group.

   For more information about using the `group` component, see Section 8.6.2, "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.

   Facets accept only one child component. If you have to insert more than one component in the `footer` facet, use the `panelGroupLayout` component or the `group` component to wrap the `footer` child components. Example 8–5 shows sample code that uses the `panelGroupLayout` component to arrange `footer` child components in a `panelFormLayout` component.

Example 8–5 Footer Child Components in panelFormLayout Arranged Horizontally

```
<af:panelFormLayout>
  <f:facet name="footer">
    <af:panelGroupLayout layout="horizontal">
      <af:commandButton text="Save"/>
      <af:commandButton text="Cancel"/>
      <f:facet name="separator">
        <af:spacer width="3" height="3"/>
      </f:facet>
    </af:panelGroupLayout>
  </f:facet>
  .
  .
  .
</af:panelFormLayout>
```

8.6.2 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 as a child in the `panelFormLayout` component, visible separators are 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. Example 8–15 shows sample code that groups two sets of child components inside a `panelFormLayout` component.

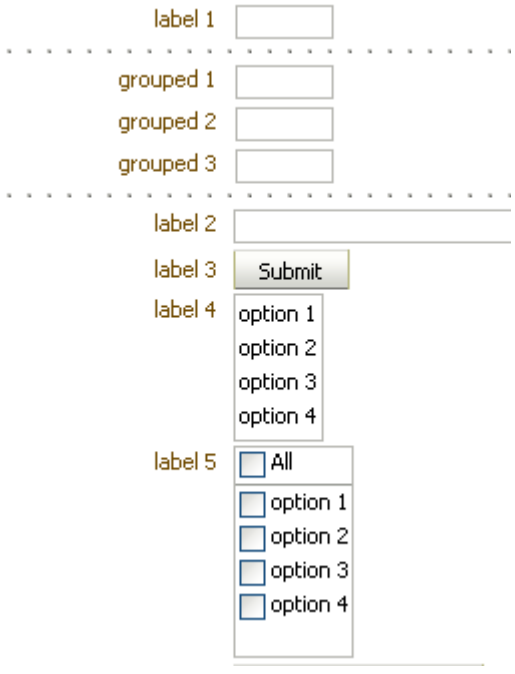
Example 8–6 Grouping Child Components in panelFormLayout

```
<af:panelFormLayout binding="#{editor.component}" rows="10" labelWidth="33%"
```

```
                    fieldWidth="67%" testId="panelFormLayout1">
  <af:inputText columns="5" label="label 1"/>
  <af:group>
    <af:inputText columns="5" label="grouped 1" shortDesc="This one is secret!"
         secret="true"/>
    <af:inputText columns="5" label="grouped 2"/>
    <af:inputText columns="5" label="grouped 3"/>
   </af:group>
   <af:inputDate id="df1" label="label 2"/>
   <af:panelLabelAndMessage label="label 3" labelStyle="vertical-align: middle;">
     <af:commandButton text="Submit"/>
   </af:panelLabelAndMessage>
   <af:selectOneListbox id="sol" label="label 4" shortDesc="Select One Option">
     <af:selectItem label="option 1"/>
     <af:selectItem label="option 2"/>
     <af:selectItem label="option 3"/>
     <af:selectItem label="option 4"/>
   </af:selectOneListbox>
   <af:selectManyListbox id="rs" label="label 5" shortDesc="Select Option">
     <af:selectItem label="option 1"/>
     <af:selectItem label="option 2"/>
     <af:selectItem label="option 3"/>
     <af:selectItem label="option 4"/>oiiiik,
     </af:selectManyListbox>
</af:panelFormLayout>
```

Following along with the sample code in Example 8–15, at runtime the `panelFormLayout` component renders dotted, separator lines before and after the first `group` of child components, as shown in Figure 8–16.

Figure 8–16 Grouped Components in panelFormLayout



As described in Section 8.6, "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.

While the `group` component does not provide any layout for its child components, the underlying HTML elements can provide the desired layout for the child components inside the `group` component. For example, if you want child button components in a `group` component to flow horizontally in a form layout, use the `panelGroupLayout` component to wrap the buttons, and set the `layout` attribute on `panelGroupLayout` component to `horizontal`. Then insert the `panelGroupLayout` component into `group` component, as shown in Example 8–7.

Example 8–7 panelGroupLayout Inside a Group Component

```
<af:group>
  <af:panelGroupLayout layout="horizontal">
    <af:commandButton text="Save" ../>
    <af:commandButton text="Cancel" ../>
    <f:facet name="separator">
      <af:spacer width="3"/>
    </f:facet>
  </af:panelGroupLayout>
</af:group>
```

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 8–8.

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

```
<af:panelFormLayout ...>
  <f:facet name="footer">
    <!-- One root group component needed -->
    <af:group>
      <af:outputText value="Footer item 1"/>
      <!-- One group -->
      <af:group>
        <af:outputText value="Group 1 item 1"/>
        <af:outputText value="Group 1 item 2"/>
      </af:group>
      <af:panelGroupLayout layout="horizontal">
        <af:commandButton text="Save"/>
        <af:commandButton text="Cancel"/>
        <f:facet name="separator">
          <af:spacer width="3"/>
        </f:facet>
      </af:panelGroupLayout>
    </af:group>
  </f:facet>
  .
  .
  .
</af:panelFormLayout>
```

Like grouped child components in a `panelFormLayout` component, at runtime the `panelFormLayout` component renders dotted, separator lines around the child components of each `group` component in the `footer` facet, as shown in Figure 8–17.

Figure 8–17 Footer in panelGroupLayout with Grouped Components

Note: 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>
    <af:outputText value="Footer item 1"/>
    <!-- Any number of groups at this level -->
    <af:group>
      <af:outputText value="Group 1 item 1"/>
      <af:outputText value="Group 1 item 2"/>
      <!-- But not another nested group. This is illegal. -->
      <af:group>
        <af:outputText value="Nested Group 1 item 1"/>
        <af:outputText value="Nested Group 1 item 2"/>
      </af:group>
    </af:group>
    <af:outputText value="Another footer item"/>
  </af:group>
</f:facet>
```

Whether you are grouping components in the `footer` facet or in the main body of the `panelFormLayout` component, if the first or last child inside the `panelFormLayout` component or inside the `footer` facet is a `group` component, no separator lines will be displayed around the child components in that group. For example, both sets of code examples in Example 8–9 would produce the same visual effect at runtime.

Example 8–9 Code Producing Same Visual Effect

```
<!-- Example 1: Group of buttons is last child in root group -->
<f:facet name="footer">
  <af:group>
    <af:outputText value="Footer text item 1"/>
    <af:outputText value="Footer text item 2"/>
    <af:group>
      <af:inputText label="Nested group item 1"/>
      <af:inputText label="Nested group item 2"/>
```

```
    </af:group>
    <af:group>
      <af:panelGroupLayout layout="horizontal">
        <af:commandButton text="Cancel"/>
        <af:commandButton text="Save"/>
      </af:panelGroupLayout>
    </af:group>
  </af:group>
</f:facet>

<!-- Example 2: panelGroupLayout of buttons is last child in root group-->
<f:facet name="footer">
  <af:group>
    <af:outputText value="Footer text item 1"/>
    <af:outputText value="Footer text item 2"/>
    <af:group>
      <af:inputText label="Nested group item 1"/>
      <af:inputText label="Nested group item 2"/>
    </af:group>
    <af:panelGroupLayout layout="horizontal">
      <af:commandButton text="Cancel"/>
      <af:commandButton text="Save"/>
    </af:panelGroupLayout>
  </af:group>
</f:facet>
```

8.7 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 8–18.

Figure 8–18 panelDashboard with panelBox Child Components

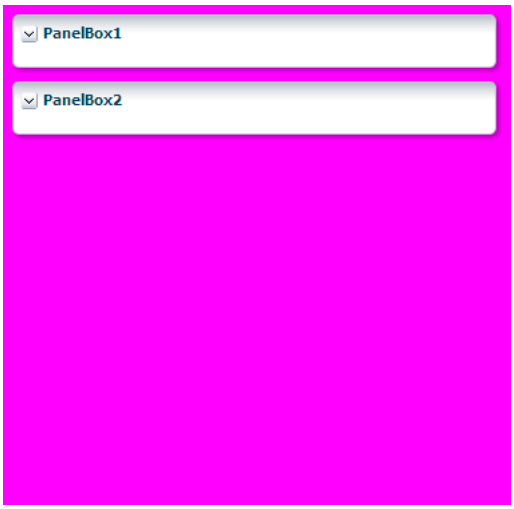
When you add a `panelDashboard` component, you configure the number of columns it will contain, along with the height of each row. The dashboard stretches its children to fill up the configured space. If all the child components do not fit within the

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 8–19 (the color of the dashboard has been changed to fuchsia to make it more easy to see its boundaries).

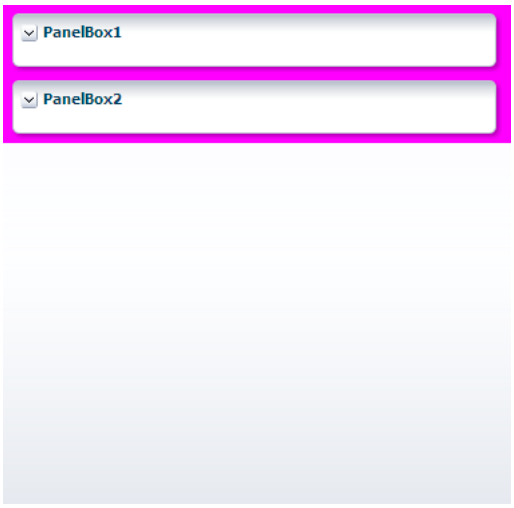
Figure 8–19 panelDashboard Stretches to Fill Space



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 8–20.

Figure 8–20 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 8–21, 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 8–21 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 32.6, "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.

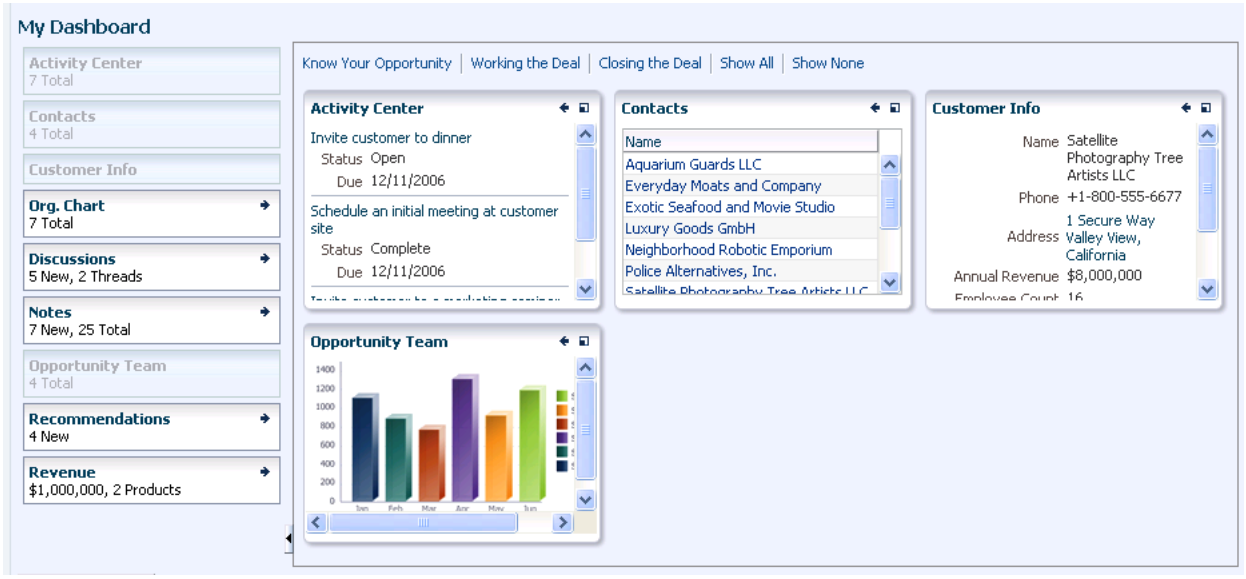
For example, Figure 8–22 shows a `panelDashboard` component used in the right panel of a `panelSplitter` component. In the left panel, list items displayed as links represent each `panelBox` component in the `panelDashboard`. When all `panelBox` components are displayed, the links are all inactive. However, 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 8–22 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 8–23 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 8–23 Practical Example of panelDashboard



8.7.1 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.

To use the panelDashboard component:

1. In the Component Palette, from the Layout panel drag and drop a **Panel Dashboard** onto the page.
2. In the Property Inspector, 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. To do so, expand the **Other** 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 Component Palette, 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 Component Palette, from the Operations panel, 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 32, "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 Property Inspector, 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 8–10 shows code on a managed bean that handles the insertion of child components.

Example 8–10 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 Component Palette, from the Operations panel, drag a **Panel Dashboard Behavior** tag and drop it as a child to the command component.
13. In the Property Inspector, 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.

8.7.2 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`
- `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.

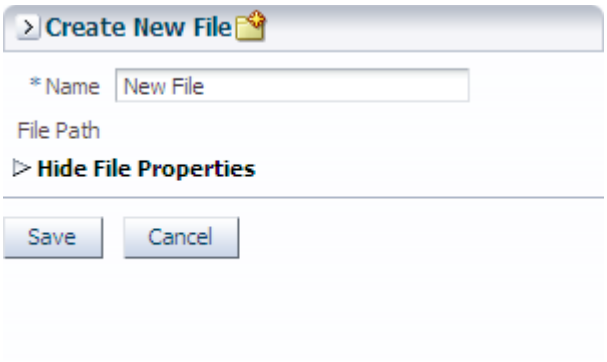
8.8 Displaying and Hiding Contents Dynamically

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 toggle icon is a plus sign in a box. When the contents are disclosed (shown), the default label is **Hide**, and the toggle icon changes to a minus sign.

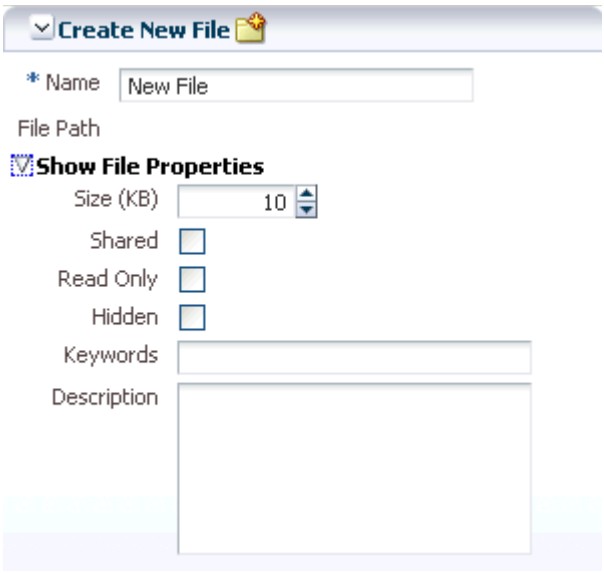
For example, the `newFileItem` page of the File Explorer application uses a `showDetail` component to hide and display file properties. The component is configured to hide the properties when the page is displayed, as shown in Figure 8–24.

Figure 8–24 Collapsed showDetail



When the user clicks the toggle icon, the properties are displayed, as shown in Figure 8–25.

Figure 8–25 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 8.8.1, "How to Use the showDetail Component."

> **Tip:** By default, child components of the `showDetail` component are indented. You can control the indentation using the `child-container` skinning key. For example:
>
> ```
> af|showDetail {
>   -tr-layout: flush;
> }
> af|showDetail::child-container {
>   padding-left: 10px;
> }
> ```
>
> For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

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 8.10, "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. When the user hovers over the truncated text, the full text is displayed in a tooltip, as shown in Figure 8–26.

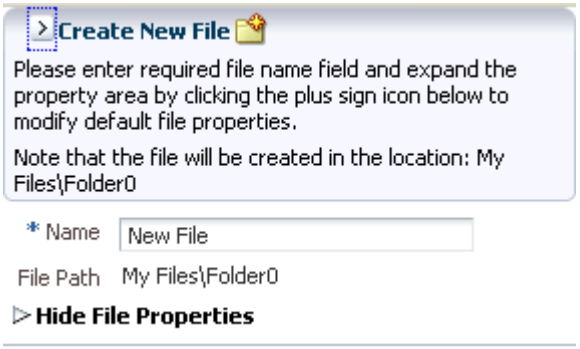
Figure 8–26 Text for the showDetailHeader Is Truncated

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 8–27.

Figure 8–27 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 hidden or displayed 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 not displayed, as shown in Figure 8–25. When the user clicks the toggle icon in the header, the contents are displayed, as shown in Figure 8–28.

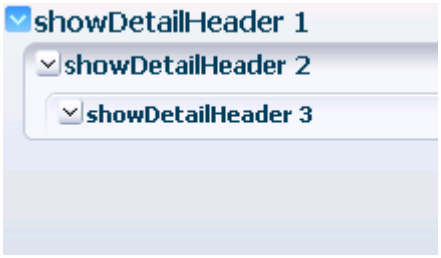
Figure 8–28 showDetailHeader Component Used to Display Help



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 8.10, "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 8–29 shows three nested `showDetailHeader` components, and their different styles.

Figure 8–29 Nested showDetailHeader Components Create a Hierarchy



You can change the styles used by each header level by applying a skin to the `showDetailHeader` component. For details about skinning ADF Faces components, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

> **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 8–30.

Figure 8–30 Two panelBox Components

Figure 8–31 shows the same page, but with the History `panelBox` component in an undisclosed state.

Figure 8–31 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 8–32 shows the same panel boxes as in Figure 8–30, but with the bottom `panelBox` component configured to show the medium tone of the core ramp.

Figure 8–32 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.

If you want to show and hide multiple large areas of content, consider using the `panelAccordion` and `panelTabbed` components. For more information, see Section 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed Panels."

8.8.1 How to Use the showDetail Component

Use the `showDetail` component to show and hide a single set of content.

To create and use the showDetail component:

1. In the Component Palette, from the Common Components panel, drag and drop a **Show Detail** from the Component Palette onto the JSF page.

Tip: This component appears in the Common Components panel of the Component Palette, and not the Layout panel.

2. In the Property Inspector, 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 31, "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.

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 8.8.4, "What You May Need to Know About Disclosure Events."

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

8.8.2 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.

To create and use the showDetailHeader component:

1. In the Component Palette, from the Layout panel, drag and drop a **Show Detail Header** onto the JSF page.

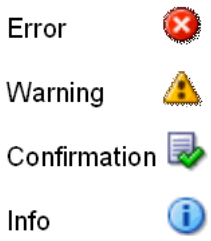
2. In the Property Inspector, expand the Common section. Set **Text** to the text string you want for the section header label.
3. Set **Icon** to the URI of the image file you want to use for the section header icon. The icon image is displayed before the header label.

   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 8–33 shows each of the icons used for message types.

Figure 8–33 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 31, "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 8.8.4, "What You May Need to Know About Disclosure Events."

7. If you want to control how the `showDetailHeader` component handles geometry management, expand the Other 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 8.2.1, "Geometry Management and Component Stretching."

8. To add buttons or icons to the header, in the Component Palette, from the Common Components panel, drag and drop the `toolbar` component into the `toolbar` facet. Then add any number of `commandToolbarButton` or `commandButton` components into the newly inserted `toolbar` component. For more information about using the `toolbar` component, see Section 14.3, "Using Toolbars."

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

9. To add menus to the header, insert menu components into the `menuBar` facet. For more information about creating menus, see Section 14.2, "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.

10. 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.

11. 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. 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 20, "Customizing the Appearance Using Styles and Skins."

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

8.8.3 How to Use the panelBox Component

You can insert any number of `panelBox` components on a page.

To create and use a panelBox component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Box** to the JSF page.
2. In the Property Inspector, 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 details, see Chapter 20, "Customizing the Appearance Using Styles and Skins."
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 8.8.4, "What You May Need to Know About Disclosure Events."
8. To add toolbar buttons, in the Component Palette, from the Common Components Panel, drag and drop a **Toolbar** into the `toolbar` facet. Then insert the desired number of `commandToolbarButton` components into the `toolbar` component. For information about using `toolbar` and `commandToolbarButton` components, see Section 14.3, "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.
9. 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.
10. 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 8–11 shows the code you might use for changing the width.

Example 8–11 panelBox Component with inlineStyle Attribute Set

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

8.8.4 What You May Need to Know About Disclosure Events

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.

The `disclosed` attribute on these components 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.

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 31, "Allowing User Customization on JSF Pages."

When the user clicks the toggle icon to show or hide contents, 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 8–12.

Example 8–12 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.

On the client-side component, the `AdfDisclosureEvent` event is fired. 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 5, "Handling Events."

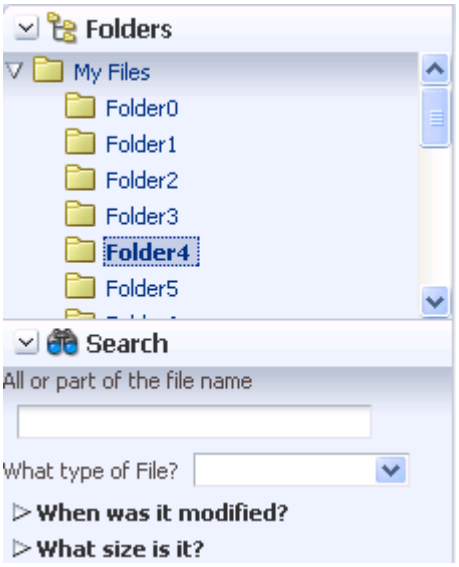
8.9 Displaying or Hiding Contents in Accordion Panels and Tabbed Panels

When you need to display multiple areas of content that can be hidden and displayed, you can use the `panelAccordion` or the `panelTabbed` components. Both of 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.

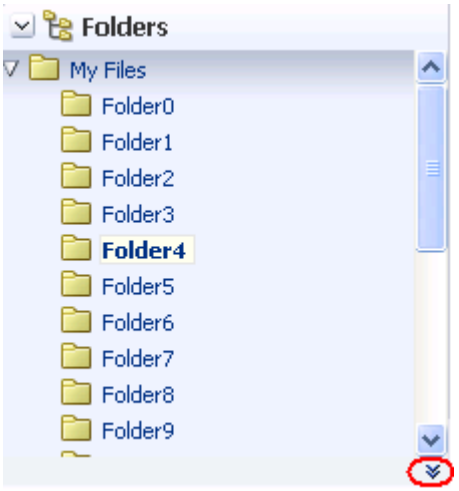
When a panel is collapsed, only the panel header is displayed; when a panel is expanded, the panel contents are displayed beneath the panel header (users can expand the panes by clicking either the `panelAccordion` component's header or the expand icon). The File Explorer application uses the `panelAccordion` component to display the Folders and Search panes, as shown in Figure 8–34.

Figure 8–34 panelAccordion Panes



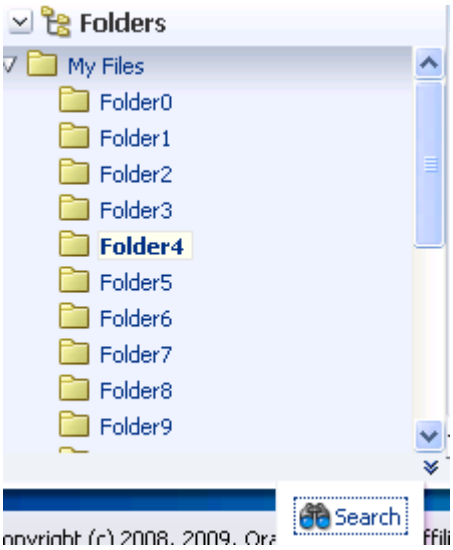
At runtime, when available browser space is less than the space needed to display expanded panel contents, ADF Faces automatically displays overflow icons that enable users to select and navigate to those panes that are out of view. Figure 8–35 shows the overflow icon (circled in the lower right-hand corner) displayed in the Folders panel of the File Explorer application when there is not enough room to display the Search panel.

Figure 8–35 Overflow Icon In panelAccordion



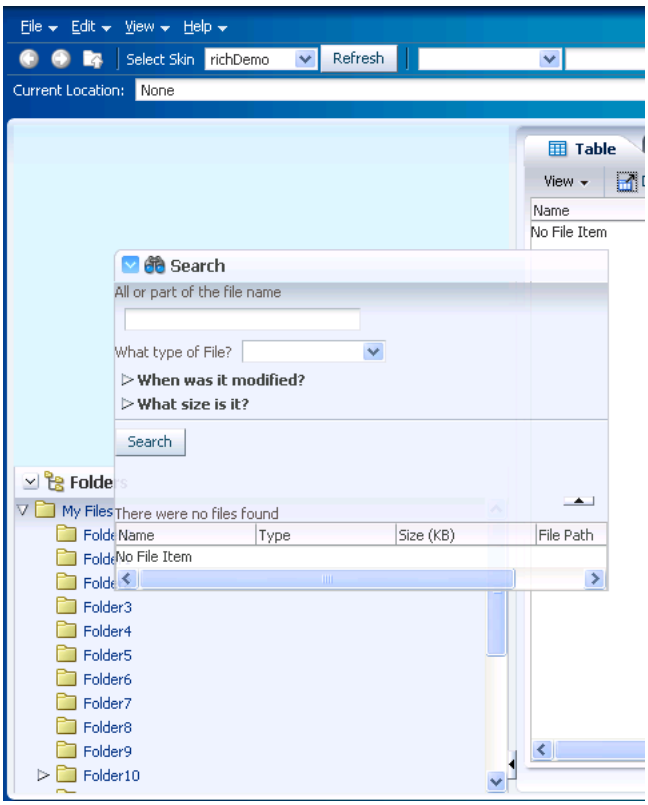
When the user clicks the overflow icon, ADF Faces displays the overflow popup menu (as shown in Figure 8–36) for the user to select and navigate to.

Figure 8–36 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 8–37.

Figure 8–37 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.

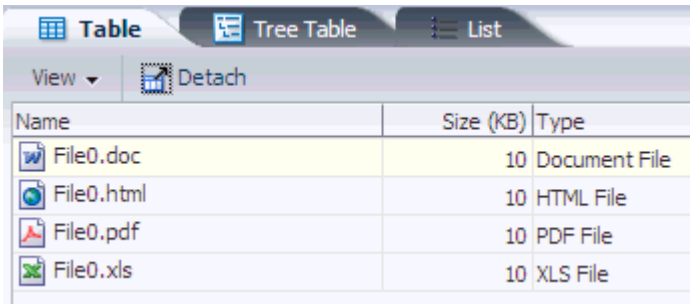
To use the `panelAccordion` component, see Section 8.9.1, "How to Use the panelAccordion Component."

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 the display area, below the display area, or both. You can configure a `panelTabbed` component so that the individual tabs can be closed. You can have it so that all tabs can be closed, all but the last tab can be closed, or no tabs can be closed. When tabs are configured to be closed, an X is displayed at the end of the tab. You can also configure tabs so that they display a disabled X, meaning it can be closed, but is currently disabled.

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 8–38.

Figure 8–38 panelTabbed Panes



To use the `panelTabbed` component, see Section 8.9.2, "How to Use the panelTabbed Component."

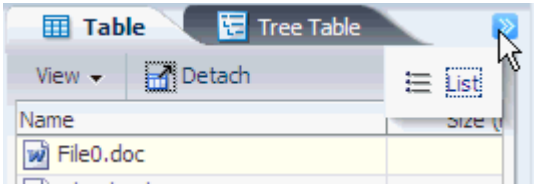
> **Tip:** If you want the tabs to be used in conjunction with navigational hierarchy, for example, each tab is a different page or region that contains another set of navigation items, you may want to use a navigation panel component to create a navigational menu. For more information, see Section 18.5, "Using Navigation Items for a Page Hierarchy."

For both the `panelAccordion` and `panelTabbed` components, use one `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` or `panelTabbed` components, respectively. To use the `showDetailItem` component, see Section 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components." You

can add a toolbar to the `toolbar` facet of the `showDetailItem` component, and the toolbar will be shown whenever the panel or tab is disclosed. Figure 8–38 shows the toolbar used by the `showDetailItem` component in the File Explorer application.

The `panelTabbed` component also supports an overflow icon if all tabs cannot be displayed. Figure 8–39 shows the overflow icon in the File Explorer application.

Figure 8–39 Overflow Icon in panelTabbed Component



> **Performance Tip:** The number of child components within a `panelAccordion` or `panelTabbed` component, and the complexity of the child components, will affect the performance of the overflow. Set the size of the `panelAccordion` or `panelTabbed` component to avoid overflow when possible.

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.

When you configure the `panelAccordion` or `panelTabbed` 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.

8.9.1 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 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components."

To create and use the panelAccordion component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Accordion** onto the JSF page.
2. In the Property Inspector, 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 Other 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. To do so, expand the **Other** section, and 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 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components."

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 **Other** 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 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components."

8.9.2 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.

To create and use the panelTabbed component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Tabbed** onto the JSF page.
2. In the Property Inspector, expand the Common section.
3. Set **Position** to `below` if you want the tabs to be rendered below the contents in the display area.

   By default, the value is `above`. This means the tabs are rendered above the contents in the display area. The other acceptable value is `both`, where tabs are rendered above and below the display area.

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 showDetail Item component, so that an individual tab cannot be removed (a close icon does not display), or so that the closed icon is disabled. For more information, see Section 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components."

5. 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. To do so, expand the **Other** section, and 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.

6. 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 8.9.3, "How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components."

8.9.3 How to Use the showDetailItem Component to Display Content in panelAccordion or panelTabbed Components

Insert `showDetailItem` components into a `panelAccordion` or `panelTabbed` component only. Each `showDetailItem` component corresponds to one accordion panel or tabbed 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 accordion panel or tab 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` or `panelTabbed` component to the JSF page, as described in Section 8.9.1, "How to Use the panelAccordion Component," and Section 8.9.2, "How to Use the panelTabbed Component," respectively.

To add accordion panel or tabbed panel contents using a showDetailItem component:

1. Insert one or more `showDetailItem` components inside the parent component, such as `panelAccordion` or `panelTabbed`, by dragging and dropping a **Show Detail Item** component from Common Components panel of the Component Palette.
2. In the Property Inspector, expand the **Appearance** section.
3. Set **Text** to the label you want to display for this panel or tab.
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 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.

   Note: If you have set the `panelAccordion` to not stretch (that is, you've set `dimensionsFrom` to `children`), then you cannot set values for the `flex` and `stretchChildren` attributes, as it will result in a circular reference back to the `panelAccordion` for size.

   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 8–36). 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 8.9.4, "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 31, "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 8.9.4, "What You May Need to Know About Geometry Management and the showDetailItem Component."

6. Expand the Behavior section. Set **DisclosureListener** to the `disclosureListener` method in a backing bean you want to execute when this panel or tab is selected by the user.

   For information about server disclosure events and event listeners, see Section 8.8.4, "What You May Need to Know About Disclosure Events."

7. Set **Disabled** to `true` if you want to disable this panel or tab (that is, the user will not be able to select the panel or tab).

8. Set **Disclosed** to `true` if you want this panel or tab to show its child components.

   By default, the `disclosed` attribute is set to `false`. This means the contents for this panel or tab 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 31, "Allowing User Customization on JSF Pages."

9. For `showDetailItem` components used in a `panelAccordion` component, expand the Other 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.

10. If you chose to allow tab removal for a `panelTabbed` component, expand the Other 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.
11. To add toolbar buttons to a panel (supported in the `panelAccordion` component only), in the Component Palette, from the Common Components panel, insert a **Toolbar** into the `toolbar` facet of the `showDetailItem` component that defines that panel. Then, insert the desired number of `commandToolbarButton` 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 `commandToolbarButton`, see Section 14.3, "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.
12. To add contents to the panel, insert the desired child components into each `showDetailItem` component.

8.9.4 What You May Need to Know About Geometry Management and the showDetailItem Component

Both the `panelAccordion` or `panelTabbed` components can be configured to stretch when they are placed inside a component that uses geometry management to stretch its child components. However, 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`. By default, panel contents will not stretch.

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:

- `inputText` (when configured to stretch)
- `decorativeBox` (when configured to stretch)
- `panelAccordion` (when configured to stretch)
- `panelBox`
- `panelCollection`
- `panelDashboard` (when configured to stretch)
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (only when the `layout` attribute is set to `scroll` or `vertical`)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)
- `region`
- `table` (when configured to stretch)
- `tree` (when configured to stretch)
- `treeTable` (when configured to stretch)

The following components cannot be stretched when placed inside a `showDetailItem` component:

- `panelBorderLayout`
- `panelFormLayout`
- `panelGroupLayout` (only when the `layout` attribute is set to `default` or `horizontal`)
- `panelHeader`
- `panelLabelAndMessage`
- `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 8.2.1, "Geometry Management and Component Stretching."

8.9.5 What You May Need to Know About showDetailItem Disclosure Events

The `showDetailItem` component inside of `panelAccordion` and `panelTabbed` 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 `panelTabbed` or `panelAccordion` component, the event root is the `panelTabbed` or `panelAccordion` 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` or `panelAccordion` 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 8.8.4, "What You May Need to Know About Disclosure Events."

8.10 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.

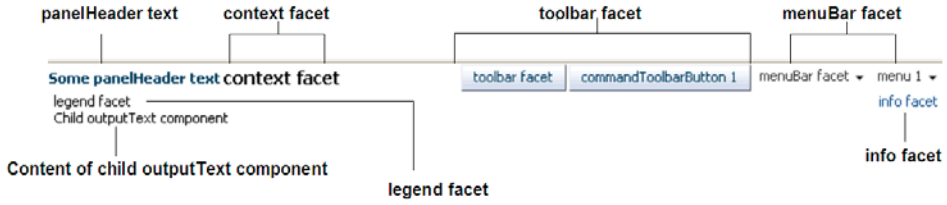
You can use the `decorativeBox` component when you need to transition to a different look and feel on the page. The `decorativeBox` component uses themes and skinning keys to control the borders and colors of its different facets. For example, depending on the skin you are using, if you use the default 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. For information about using themes and skins, see Chapter 20, "Customizing the Appearance Using Styles and Skins"

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 8–40 shows the different facets in the `panelHeader` component.

Figure 8–40 panelHeader and Its Facets

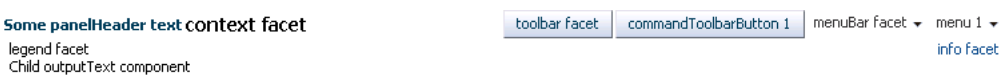


When there is not enough space to display everything in all the facets of the title line, the `panelHeader` text is truncated and displays an ellipsis. When the user hovers over the truncated text, the full text is displayed in a tooltip, as shown in Figure 8–41.

Figure 8–41 Text for the panelHeader Is Truncated

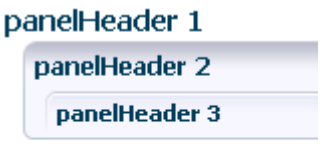
When there is more than enough room to display the contents, the extra space is placed between the `context` facet and the toolbar, as shown in Figure 8–42.

Figure 8–42 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 8–43, you can have a main header with a subheader and then a heading level 1 also with a subheader.

Figure 8–43 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.

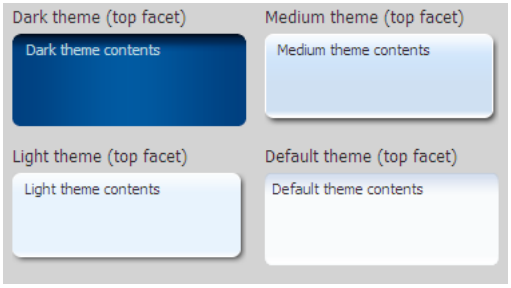
> **Note:** Heading sizes are determined by default by the physical containment of the header components. That is, the first header component will render as a heading level 1. Any header component nested in the first header component will render as a heading level 2, and so on. You can manually override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

For information about using the `panelHeader` component, see Section 8.10.1, "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 non-colored 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.

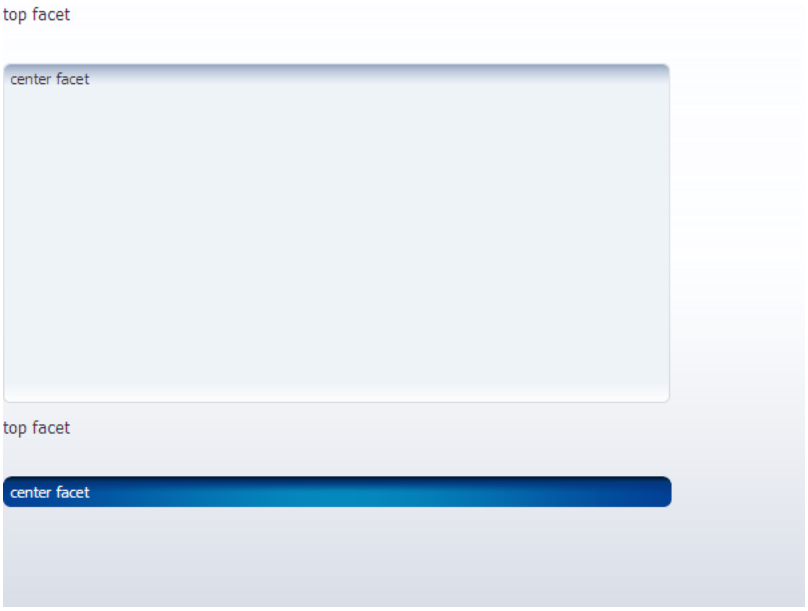
The color of the box for the center facet depends on the theme and skin used. Figure 8–44 shows the different themes available by default.

Figure 8–44 Themes Used in a decorativeBox Component



By default, the `decorativeBox` component stretches to fill its parent component. You can also configure the `decorative` Box component to inherit its dimensions from its child components. For example, Figure 8–45 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 8–45 decorativeBox Can Stretch or Not



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 20, "Customizing the Appearance Using Styles and Skins."

8.10.1 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 8.8.2, "How to Use the showDetailHeader Component."

To create and use a panelHeader component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Header** onto the page.
2. In the Property Inspector, 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 8–46 shows the icons used for the different message types.

Figure 8–46 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 17.5, "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. 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 20, "Customizing the Appearance Using Styles and Skins."

9. If you want to control how the `panelHeader` component handles geometry management, expand the Other section and set **Type** to one of the following. For more information about geometry management, see Section 8.2.1, "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 `commandToolbarButton` components into the `toolbar` component. For information about using `toolbar` and `commandToolbarButton`, see Section 14.3, "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 14.2, "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.

8.10.2 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 18.5, "Using Navigation Items for a Page Hierarchy."

To create and use a decorativeBox component:

1. In the Component Palette, from the Layout panel, drag and drop a **Decorative Box** onto the page.
2. In the Property Inspector, 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. To do so, expand the **Other** section, and 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 8.10.3, "What You May Need to Know About Geometry Management and the decorativeBox Component."

8.10.3 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`
- `panelDashboard`
- `panelGridLayout` (when `gridRow` and `gridCell` components are configured to stretch)
- `panelGroupLayout` (only with the `layout` attribute set to `scroll` or `vertical`)
- `panelSplitter` (when configured to stretch)
- `panelStretchLayout` (when configured to stretch)
- `panelTabbed` (when configured to stretch)

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

The following components cannot be stretched when placed inside a facet of the decorativeBox component:

- panelBorderLayout
- panelFormLayout
- panelGroupLayout (only with the layout attribute set to default or horizontal)
- panelHeader
- panelLabelAndMessage
- panelList
- showDetail
- showDetailHeader
- tableLayout (MyFaces Trinidad component)

You cannot place components that cannot stretch 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.

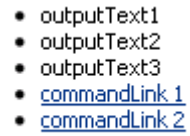
For example, if you want to place content in a panelBox component and have it flow within a facet of the decorativeBox component, you could place a panelGroupLayout component with its layout attribute set to scroll in the facet of the decorativeBox component, and then place the panelBox component in that panelGroupLayout component. For more information, see Section 8.2.2, "Nesting Components Inside Components That Allow Stretching."

8.11 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 8–47. 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 9.6, "Using Selection Components." If you need to create lists that change the model layer, see Chapter 11, "Using List-of-Values Components."

Figure 8–47 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.

8.11.1 How to Use the panelList Component

Use one `panelList` component to create each list of items.

To create and use the panelList component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel List** to the JSF page.
2. In the Property Inspector, 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`

   Example 8–13 shows the code for setting the list style to a circle.

Example 8–13 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.

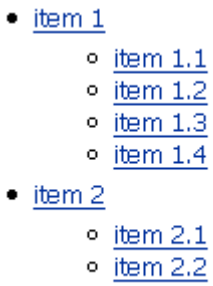
   For example, you could insert a series of `commandLink` components or `outputFormatted` components.

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 8.6, "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.

8.11.2 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 8–48, 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 8–48 Hierarchical List Created Using Nested panelList Components



To achieve the list hierarchy as shown in Figure 8–48, use a `group` component to wrap the components that make up each group of outer items and their respective inner items. Example 8–14 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 8–14 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 8.12, "Grouping Related Items."

8.12 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.

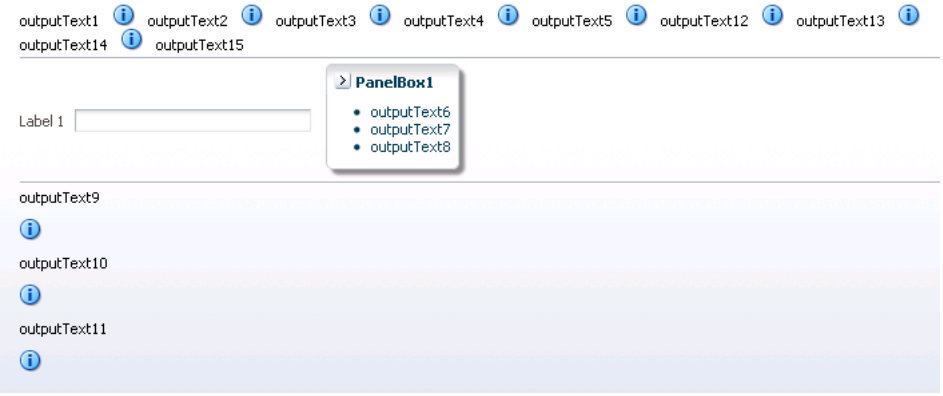
You can use any number of `group` components to group related components together. For example, you might want to group some of the input fields in a form layout created by the `panelFormLayout` component. Example 8–15 shows sample code that groups two sets of child components inside a `panelFormLayout` component.

Example 8–15 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 8–49. The `layout` attribute value determines the arrangement of the child components.

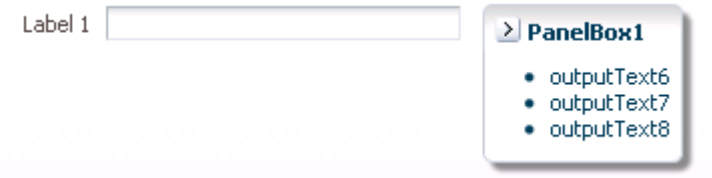
Figure 8–49 panelGroupLayout Arrangements



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 8.13, "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 8–50.

Figure 8–50 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.

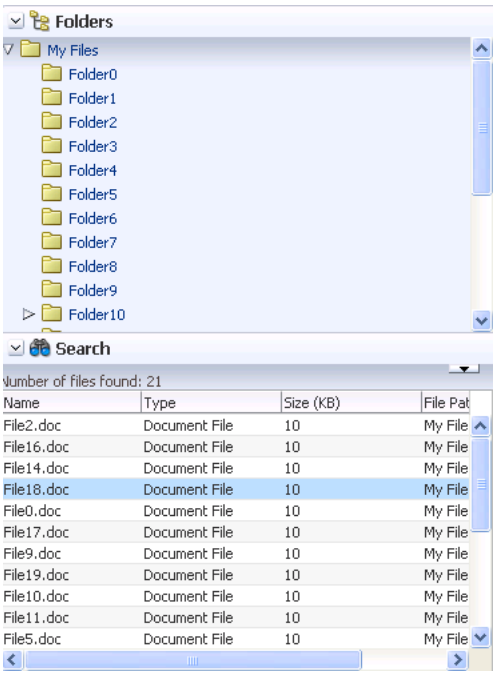
When the `layout` attribute is set to `scroll` on a `panelGroupLayout` component, ADF Faces automatically provides a scrollbar at runtime when the contents contained by the `panelGroupLayout` component are larger than the `panelGroupLayout` component itself. You do not have to write any code to enable 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 8–51.

Figure 8–51 Scrollbars Rendered Using panelGroupLayout



8.12.1 How to Use the panelGroupLayout Component

Any number of `panelGroupLayout` components can be nested to achieve the desired layout.

To create and use the panelGroupLayout component:

1. In the Component Palette, from the Layout panel, drag and drop a **Panel Group Layout** to 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 Property Inspector, 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:

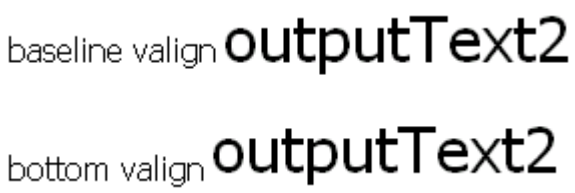
  – **halign**: Sets the horizontal alignment. The default is `center`. Other acceptable values are: `start`, `end`, `left`, `right`.

    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 8–52. 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 8–52 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.

8.12.2 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`.

8.13 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.

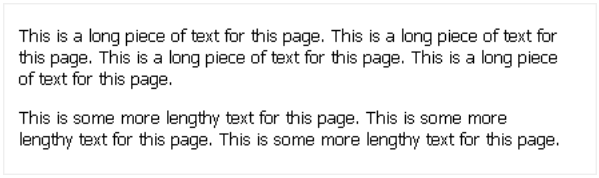
The `height` attribute determines the amount of vertical space to include in the page. Example 8–16 shows a page set up to space out two lengthy `outputText` components with some vertical space.

Example 8–16 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 8–53 shows the effect the `spacer` component has on the page output as viewed in a browser.

Figure 8–53 Vertical Space Viewed in a Browser



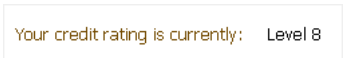
The `width` attribute determines the amount of horizontal space to include between components. Example 8–17 shows part of the source of a page set up to space out two components horizontally.

Example 8–17 Horizontal Space

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

Figure 8–54 shows the effect of spacing components horizontally as viewed in a browser.

Figure 8–54 Horizontal Space Viewed in a Browser



The `separator` component creates a horizontal line. Figure 8–55 shows the `properties.jspx` file as it would be displayed with a `separator` component inserted between the two `panelBox` components.

Figure 8–55 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.

8.13.1 How to Use the spacer Component

You can use as many `spacer` components as needed on a page.

To create and use the spacer component:

1. In the Component Palette, from the Layout panel, drag and drop a **Spacer** to the JSF page.
2. In the Property Inspector, 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.

8.13.2 How to Use the Separator Component

You can use as many `separator` components as needed on a page.

To create and use the separator component:

1. In the Component Palette, from the Layout panel, drag and drop a **Separator** to the JSF page.
2. In the Property Inspector, set the properties as needed.

9

Using Input Components and Defining Forms

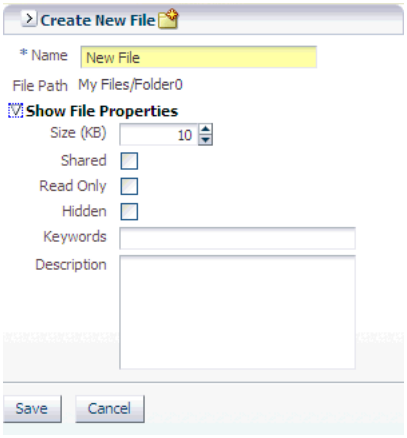
This chapter describes the input components that are used to enter data, select values, edit text, and load files.

This chapter includes the following sections:

9.1 Introduction to 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. For example, the File Explorer application contains a form that allows users to create a new file. Using input components, users enter the name, the size, select permissions, and add keywords, and a description, as shown in Figure 9–1.

Figure 9–1 Form Uses Input Components



In addition to standard input components used to input text, number, date, or color, ADF Faces includes input type components that provide additional functionality. The `inputFile` component allows users to browse for a file to load.

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. The `richTextEditor` component can also be used with command components to insert given text into the component. The inserted text can be preformatted. Additionally, you can customize the buttons that appear in the editor's toolbar.

The selection components allow the user to make selections from a list of items instead of or in addition to typing in values. For example, the `selectOneChoice` component lets the user select input from a dropdown list and the `selectOneRadio` component lets a user pick from a group of radio buttons.

You can use either selection 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. For more information about using LOV components, see Chapter 11, "Using List-of-Values Components."

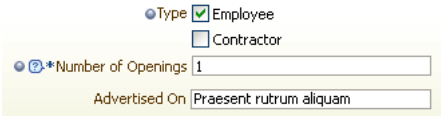
The `selectItem` component is used within other selection components to represent the individual selectable items for that component. For example, a `selectOneRadio` component will have a `selectItem` component for each of its radio buttons. If the radio button selections are coffee, tea, and milk, there would be a `selectItem` component for coffee, one for tea, and one for milk.

The form components provide a container for other components. The `form` component represents a region where values from embedded input components can be submitted. Form components cannot be nested. However, the `subform` component provides additional flexibility by defining subregions whose component values can be submitted separately within a form. The `resetButton` component provides an easy way for the user to reset input values within a form or subform to their previous state.

All the input and selection components deliver the `ValueChangeEvent` and `AttributeChangeEvent` events. You can create `valueChangeListener` and `attributeChangeListener` methods to provide functionality in response to the corresponding events.

All input components, selection components (except `selectItem`), and the rich text editor component have a `changed` attribute that when set to `true` enables a change indicator icon to be displayed upon changes in the `value` field. This indicator allows the user to easily see which input value has changed, which can be helpful when there are multiple components on the page. By default, the change indicator usually is displayed to the left of the component. If the value in a field automatically changes due to a change in another field's value, such as an automatically generated postal code when the city is entered, the postal code field will also display a change indicator. Figure 9–2 shows changed indicators present for the checkbox and input components.

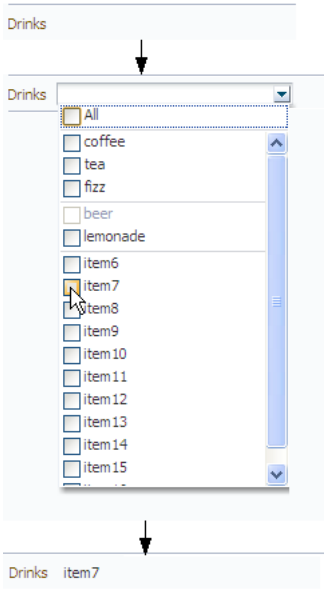
Figure 9–2 Changed indicators for two components



> **Tip:** You can change the icon or the position of the icon using skins. For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

Most input components also have the capability of displaying only the label, and not appearing capable of changing value until the user mouses over or hovers over the component. Once the user changes the value, that new value displays as read-only. Figure 9–3 shows a `selectManyChoice` component configured to be editable only on access.

Figure 9–3 Input Components Can Appear Not Editable



Input components can also display tooltips, error and validation messages, and context-sensitive help. For more information, see Chapter 17, "Displaying Tips, Messages, and Help."

All input components have JavaScript client APIs that you can use to set or get property values. For more information, see the ADF Faces JavaScript API documentation.

9.2 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 9–1 shows two input components and a Submit button that when clicked will submit both input values for processing.

Example 9–1 ADF Faces Form as a Container for Input Components

```
<af:form>
  <af:panelFormLayout>
    <af:inputText value="#{myBean.firstName}"
                  label="#{First Name}"
    </af:inputText>
    <af:inputText value="#{myBean.lastName}"
                  label="#{Last Name}"
    </af:inputText>
    <f:facet name="footer">
      <af:commandButton text="Submit"/>
    </f:facet>
   </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 4.5, "Using Subforms to Create Regions on a Page."

Example 9–2 shows a form with two subforms, each containing its own input components and Submit button. When a Submit button is clicked, only the input values within that subform will be submitted for processing.

Example 9–2 ADF Faces Subform Within a Form

```
<af:form>
  <af:subform>
    <af:panelFormLayout>
```

```
      <af:inputText value="#{myBean.firstName}"
      </af:inputText>
      <af:inputText value="#{myBean.lastName}"
      </af:inputText>
      <f:facet name="footer">
        <af:commandButton text="Submit"/>
      </f:facet>
    </af:panelFormLayout>
  </af:subform>
  <af:subform>
    <af:panelFormLayout>
      <af:inputText value="#{myBean.primaryPhone}"
      </af:inputText>
      <af:inputText value="#{myBean.cellPhone}"
      </af:inputText>
      <f:facet name="footer">
        <af:commandButton text="Submit"/>
      </f:facet>
    </af:panelFormLayout>
  </af:subform>
</af:form>
```

Aside from the basic Submit button, you can add any other command component within a form and have it operate on any field within the form. ADF Faces provides a specialized command component: the `resetButton` component, which when clicked, resets all the input and selection components within a form. That is, it updates all components whose values can be edited with the current values of the model. The `resetButton` component is different from HTML reset in that the `resetButton` component will reset the input components to their previous state which was partially or fully submitted successfully to the server without any validation or conversion error. For example, if a user enters value `A` and clicks the Submit button, and then changes the value from `A` to `B` and clicks the `resetButton` component, the value `A` will be restored.

9.2.1 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.

To add a form to a page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Form** onto the page.
2. In the Property Inspector 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. For more information about uploading files, see Section 9.9, "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`.

9.2.2 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.

To add subforms to a page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Subform** onto the page as a child to a `form` component.
2. Use the Property Inspector to 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 4, "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.

9.2.3 How to Add a Reset Button to a Form

You can add the `resetButton` component inside a form or a subform. The reset button will act upon only those components within that form or subform.

To add a reset button to a page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Reset Button** onto the page.
2. Use the Property Inspector to set the following:
   - **Text**: Specify the textual label of the button.
   - **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.

9.3 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 to create a multiple row text input, consider using the `richTextEditor` component as described in Section 9.8, "Using the richTextEditor Component."

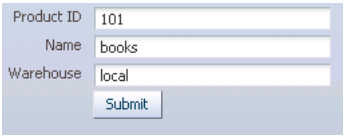
You can hide the input values from being displayed, such as for passwords, by setting the `secret` attribute to `true`. Like other ADF Faces components, the `inputText` component supports label, text, and messages. When you want this component to be displayed without a label, you set the `simple` attribute to `true`. Figure 9–4 shows a single-row `inputText` component.

Figure 9–4 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 Chapter 8.2.1, "Geometry Management and Component Stretching." Figure 9–6 shows a multi-row `inputText` component.

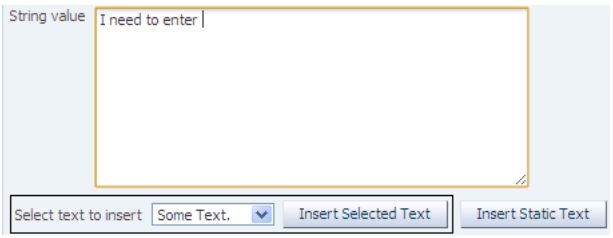
You can add multiple `inputText` components to create an input form. Figure 9–5 shows an input form using three `inputText` components and a **Submit** command button.

Figure 9–5 Form Created by inputText Components



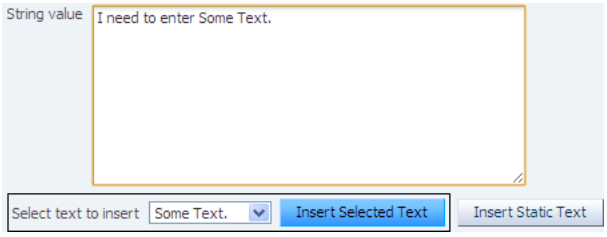
You can also configure an `insertTextBehavior` tag that works with command components to insert given text into an `inputText` component. The text to be entered can be a simple string, or it can be the value of another component, for example the selected list item in a `selectOneChoice` component. For example, Figure 9–6 shows an `inputText` component with some text already entered by a user.

Figure 9–6 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 9–7.

Figure 9–7 inputText Component with Inserted Text



9.3.1 How to Add an inputText Component

You can use an `inputText` component inside any of the layout components described in Chapter 8, "Organizing Content on Web Pages."

To add an inputText component:

1. In the Component Palette, from the Common Components panel, drag and drop an **Input Text** onto the page.
2. In the Property Inspector, 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 21, "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.
3. Expand the Appearance section, and set the following:
   - **Columns**: Specify the size of the text control by entering the maximum number of characters that can be entered into the 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 set to more than 1, you must also set the `wrap` attribute.
   - **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.
   - **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. 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 4.3, "Using the Optimized Lifecycle."
   - **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 6.3, "Adding Conversion."
   - **Validator**: Specify a method reference to a validator method using an EL expression. For more information, see Section 6.5, "Adding Validation."
5. Expand the Other section and set the following:
   - **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.

- **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.

9.3.2 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 5.4, "Sending Custom Events from the Client to the Server."

Before You Begin

Before you add an `insertTextBehavior` tag, you need to create an `inputText` component as described in Section 9.3.1, "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 18.2.1, "How to Use Command Buttons and Command Links."
2. In the Component Palette, from the Operations panel, 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 9–3 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 9–3 Using the insertTextBehavior Tag

```
<af:inputText clientComponent="true" id="idInputText" label="String value"
              value="#{demoInput.value}" rows="10" columns="60"/>
<af:selectOneChoice id="targetChoice" autoSubmit="true"
                    value="#{demoInput.choiceInsertText}"
                    label="Select text to insert">
  <af:selectItem label="Some Text." value="Some Text."/>
  <af:selectItem label="0123456789" value="0123456789"/>
  <af:selectItem label="~!@#$%^*" value="~!@#$%^*"/>
  <af:selectItem label="Two Lines" value="\\nLine 1\\nLine 2"/>
</af:selectOneChoice>
<af:commandButton text="Insert Selected Text" id="firstButton"
                  partialTriggers="targetChoice">
  <af:insertTextBehavior for="idInputText"
                         value="#{demoInput.choiceInsertText}"/>
</af:commandButton>
<af:commandButton text="Insert Static Text">
  <af:insertTextBehavior for="idInputText"
                         value="Some Static Text."/>
</commandButton>
```

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 Property Inspector.

9.4 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 9–8 in horizontal layout, and in Figure 9–9 in vertical layout.

Figure 9–8 inputNumberSlider in Horizontal Layout

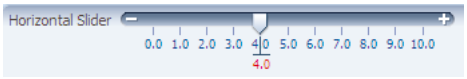
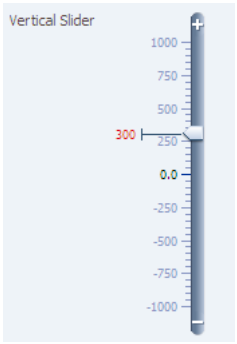


Figure 9–9 InputNumberSlider in Vertical Layout

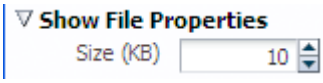


The `inputRangeSlider` component has two markers and allows the user to pick the end points of a range, as shown in Figure 9–10.

Figure 9–10 inputRangeSlider in horizontal layout

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 9–11.

Figure 9–11 inputNumberSpinbox



9.4.1 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.

To add an inputNumberSlider or inputRangeSlider component:

1. In the Component Palette, from the Common Components panel, drag and drop an **Input Number Slider** or **Input Range Slider** onto the page.
2. In the Property Inspector, 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 9–10 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 **Orientation** to specify whether the component will be in horizontal or vertical layout. For information about the other attributes in this section, see Section 9.3.1, "How to Add an inputText Component."

9.4.2 How to Add an inputNumberSpinbox Component

The `inputNumberSpinbox` component allows the user to scroll through a set of numbers to select a value.

To add an inputNumberSpinbox component:

1. In the Component Palette, from the Common Components panel, drag and drop an **Input Number Spinbox** onto the page.
2. Expand the Data section of the Property Inspector 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 section and set the attributes. For more information about setting these attributes, see Section 9.3.1, "How to Add an inputText Component."
4. If you want the value of the spinbox to appear as read-only until the user hovers over it, expand the Other 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`.

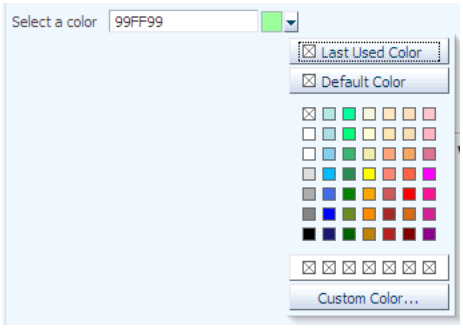
> **Note:** If you select `inherit`, and no ancestor components define the `editable` value, then the value `always` is used.

9.5 Using Color and Date Choosers

The `inputColor` component presents a text input field for entering code for colors and a button for picking colors from a palette. The default color code format is the hexadecimal color format. However, you can override the format using a `ColorConverter` class.

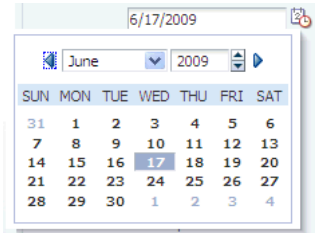
By default, the `inputColor` component opens the `chooseColor` component that allows users to pick the color from a a palette. Figure 9–12 shows the `inputColor` component with the `chooseColor` component in a popup dialog.

Figure 9–12 inputColor Component with Popup chooseColor Component



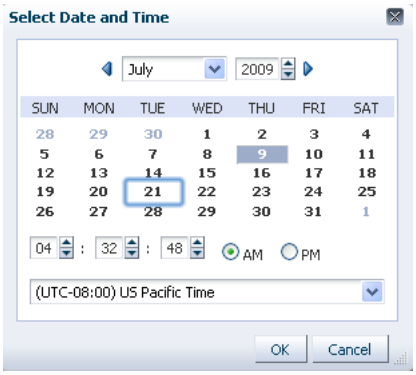
The `inputDate` component presents a text input field for entering dates and a button for picking dates from a popup calendar, as shown in Figure 9–13. The default date format is the short date format appropriate for the current locale. For example, the default format in American English (ENU) is `mm/dd/yy`. However, you can override the format using a date-time converter (for more information about using converters, see Section 6.3, "Adding Conversion").

Figure 9–13 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 timezone dropdown list is shown in the dialog, as shown in Figure 9–14.

Figure 9–14 Modal Dialog When Date-Time Converter Is Used



9.5.1 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.

To add an inputColor component:

1. In the Component Palette, from the Common Components panel, drag and drop an **Input Color** onto the page.
2. In Property Inspector, 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 9–15.

Figure 9–15 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.
   - **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. Expand the Behavior section and set the following attributes:
   - **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.
   - **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. If you want the value of the component to appear as read-only until the user hovers over it, expand the Other 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.

9.5.2 How to Add an InputDate Component

The `inputDate` component allows the user to either enter or select a date.

To add an inputDate component:

1. In the Component Palette, from the Common Components panel, drag and drop an **Input Date** onto the page.
2. In the Property Inspector, in the Common section, 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. 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.

4. Expand the Behavior section and set the following attributes:
   - **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.
   - **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.
5. 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.

9.5.3 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 timezone placeholder `z` in the converter's pattern. The end user can only modify the timezone 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 timezone 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 9–4, 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 9–4 Using an inputDate and Time Zone Selection List Together

```
<af:selectOneChoice label="Select a new timezone"
                    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);
}
```

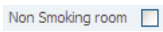
9.6 Using Selection Components

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 9–16.

Figure 9–16 selectBooleanCheckbox Component

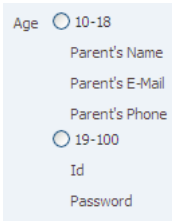


The `selectBooleanRadio` component displays a boolean choice, and must always be set to a boolean. Unlike the `selectBooleanCheckbox` component, the `selectBooleanRadio` component allows you to group `selectBooleanRadio` components together using the same `group` attribute.

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 9–17, 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.

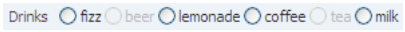
> **Tip:** Each `selectBooleanRadio` component must be bound to a unique boolean.

Figure 9–17 selectBooleanRadio Component



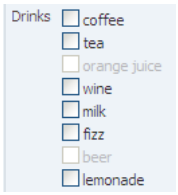
You use the `selectOneRadio` component to create a list of radio buttons from which the user can select a single value, as shown in Figure 9–18.

Figure 9–18 selectOneRadio Component



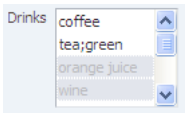
You use the `selectManyCheckbox` component to create a list of checkboxes from which the user can select one or more values, as shown in Figure 9–19.

Figure 9–19 selectManyCheckbox Component



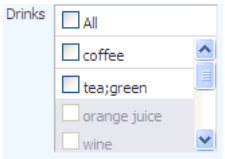
The `selectOneListbox` component creates a component which allows the user to select a single value from a list of items displayed in a shaded box, as shown in Figure 9–20.

Figure 9–20 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 9–21.

Figure 9–21 selectManyListbox Component

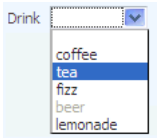


The `selectOneChoice` component creates a menu-style component, which allows the user to select a single value from a dropdown list of items. The `selectOneChoice` component is intended for a relatively small number of items in the dropdown list.

> **Best Practice:** If a large number of items is desired, use an `inputComboboxListOfValues` component instead. For more information, see Chapter 11, "Using List-of-Values Components."

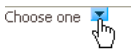
The `selectOneChoice` component is shown in Figure 9–22.

Figure 9–22 selectOneChoice Component



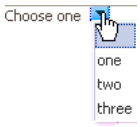
You can configure the `selectOneChoice` component to display in a compact mode, as shown in Figure 9–23. When in compact mode, the input field is replaced with a smaller icon.

Figure 9–23 selectOneChoice Component in Compact Mode



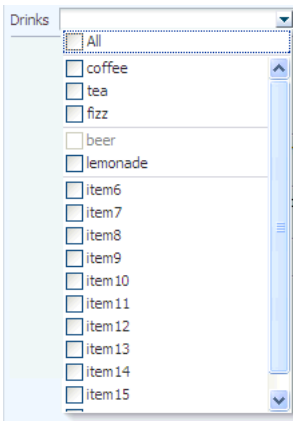
When the user clicks the icon, the dropdown list is displayed, as shown in Figure 9–24.

Figure 9–24 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 9–25.

Figure 9–25 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 9.6.2, "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 3.7.1, "How to Set Property Values on the Client"):

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

9.6.1 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.

To use a selection component:

1. In the Component Palette, from the Common Components panel, drag and drop the 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 17.3, "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 6, "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 Property Inspector and set the attributes, as described in Table 9–1. 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 9.3.1, "How to Add an inputText Component."

Table 9–1 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 Property Inspector and set the attributes, as described in Table 9–2. 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 9.3.1, "How to Add an inputText Component."

Table 9–2 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 Other 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 Other 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 9.6.2, "What You May Need to Know About the contentDelivery Attribute on the SelectManyChoice Component."

7. 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).

8. With the `selectItem` component selected, in the Property Inspector, expand the **Common** section, and if not set, enter a value for the `value` attribute. This will be the value that will be submitted.

9. Expand the Appearance section, and if not set, enter a value for **Label**. This will be the text that is displayed in the list.
10. Expand the Behavior section, and set **Disabled** to `true` if you want the item to appear disabled in the list.

9.6.2 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:

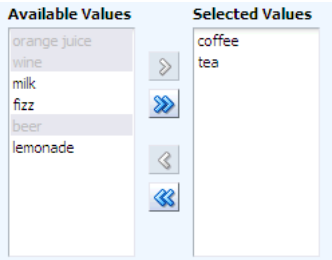
- You cannot store the value of the `selectManyChoice` is in Request scope. On postback, the value attribute is accessed from the model, rather than decoding what was returned from the client. If the value is stored in Request scope, that value will be empty. Do not store the value in Request scope.
- 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.

9.7 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.

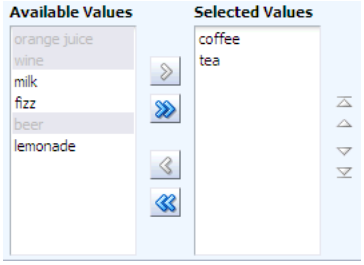
The `selectManyShuttle` component is shown in Figure 9–26.

Figure 9–26 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 9–27. 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 9–27 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.

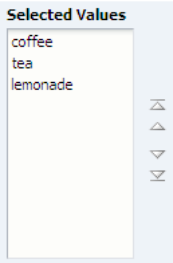
Similar to other select components, the List or Array of items are composed of `selectItem` components nested within the `selectManyShuttle` or `selectOrderShuttle` component. Example 9–5 shows a sample `selectOrderShuttle` component that allows the user to select the top five file types from a list of file types.

Example 9–5 selectOrderShuttle JSF Page Code

```
<af:selectOrderShuttle value="#{helpBean.topFive}"
    leadingHeader="#{explorerBundle['help.availableFileTypes']}"
    trailingHeader="#{explorerBundle['help.top5']}"
    simple="true">
  <af:selectItem label="XLS"/>
  <af:selectItem label="DOC"/>
  <af:selectItem label="PPT"/>
  <af:selectItem label="PDF"/>
  <af:selectItem label="Java"/>
  <af:selectItem label="JWS"/>
  <af:selectItem label="TXT"/>
  <af:selectItem label="HTML"/>
  <af:selectItem label="XML"/>
  <af:selectItem label="JS"/>
  <af:selectItem label="PNG"/>
  <af:selectItem label="BMP"/>
  <af:selectItem label="GIF"/>
  <af:selectItem label="CSS"/>
  <af:selectItem label="JPR"/>
  <af:selectItem label="JSPX"/>
  <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 9–28.

Figure 9–28 selectOrderShuttle Component in Reorder-Only Mode



9.7.1 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.

To add a selectManyShuttle or selectOrderShuttle component:

1. In the Component Palette, from the Common Components panel, drag and drop a **Select Many Shuttle** or **Select Order Shuttle** from the Component Palette 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 17.3, "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 Property Inspector, 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 Property Inspector, 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.

9.7.2 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 9–29 shows a `selectManyShuttle` component in which the user selects **Meyers** and a popup provides additional information about this selection.

Figure 9–29 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 3.2, "Listening for Client Events."

How to add a client listener to a shuttle component to handle a selection event:

1. In the Component Palette, from the Operations panel, 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 9–6.

Example 9–6 Using a clientListener to Register a Selection

```
<af:selectManyShuttle value="#{demoInput.manyListValue1}"
     valuePassThru="true" ...>
     <af:clientListener type="propertyChange" method="showDetails"/>
     <af:selectItem label="coffee" value="bean" />
      ...
</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 9–7, `AdfShuttleUtils.getLastSelectionChange` is called to get the value of the last selected item

Example 9–7 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
    }
  }
}
```

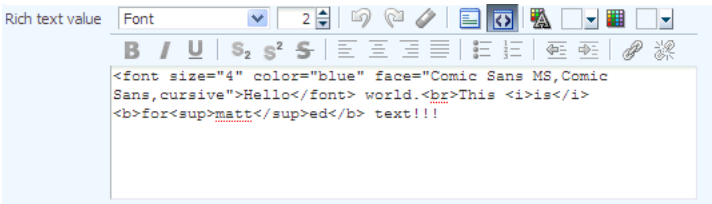
9.8 Using the richTextEditor Component

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 9–30 shows the rich text editor component in standard rich text editing Mode.

Figure 9–30 The richTextEditor Component in Standard Editing Mode

Figure 9–31 shows the editor in source code editing mode.

Figure 9–31 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 9–32.

Figure 9–32 Toolbar in richTextEditor Component

Figure 9–33 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 9–33 Customized Toolbar for richTextEditor

9.8.1 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 9.8.2, "How to Add the Ability to Insert Text into a richTextEditor Component," and Section 9.8.3, "How to Customize the Toolbar."

To add a richTextEditor component:

1. In the Component Palette, from the Common Components panel, drag and drop a **Rich Text Editor** onto the page.
2. Expand the Common section of the Property Inspector 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 10.1.1, "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 8.2.1, "Geometry Management and Component Stretching."

9.8.2 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

You need to create a `richTextEditor` component as described in Section 9.3.1, "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 18.2.1, "How to Use Command Buttons and Command Links."

2. In the Component Palette, from the Operations panel, 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 9–8 shows preformatted text as the value for an attribute in the `demoInput` managed bean.

Example 9–8 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 9–9 shows how the text is referenced from the `richTextEditorInsertBehavior` tag.

Example 9–9 Using the richTextEditorInsertBehavior Tag

```
<af:richTextEditor id="idRichTextEditor" label="Rich text value"
                   value="#{demoInput.richValue2}"/>
. . .
</af:richTextEditor>
<af:commandButton text="Insert Template Text">
  <af:richTextEditorInsertBehavior for="idRichTextEditor"
                                   value="#{demoInput.richInsertValue}"/>
</af:commandButton>
```

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.

9.8.3 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 customize the toolbar:

1. In the JSF page of the Component Palette, 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 9–33, 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 Component Palette, from the Common Components 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 14.3, "Using Toolbars."

3. With the `richTextEditor` component selected, in the Property Inspector, in the Other section, click the dropdown icon for the `toolboxLayout` attribute and select **Edit** to open the Edit Property: ToolboxLayout dialog. The value for this attribute 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`: 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.

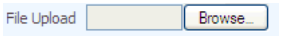
9.9 Using File Upload

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 18.4.1, "How to Use a Command 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 a file is uploaded.

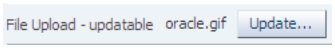
To initiate the upload process, you can create an action component such as a command button, as shown in Figure 9–34.

Figure 9–34 inputFile Component



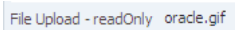
Once a file has been uploaded, and so the value of the `inputFile` is not null (either after the initial load is successful or it has been specified as an initial value), you can create an **Update** button that will be displayed instead of the **Browse** button, as shown in Figure 9–35. This will allow the user to modify the value of the `inputFile` component.

Figure 9–35 inputFile Component in Update Mode



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 9–36.

Figure 9–36 inputFile Component in Read-Only Mode



For security reasons, the following attributes cannot be set from the client:

- `disabled` (unless the `unsecure` property is set. For more information, see Section 3.7.2, "How to Unsecure the disabled Property.")
- `immediate`
- `readOnly`
- `requiredMessageDetail`
- `value`

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 9–10.

Example 9–10 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:commandButton 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 9–11.

Example 9–11 Binding the Value to a Managed Bean

```
JSF Page Code ---->
<af:form usesUpload="true">
  <af:inputFile label="Upload:" value="#{managedBean.file}" id="if1"/>
  <af:commandButton 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;

}
```

9.9.1 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.

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. In the Component Palette, from the Common Components panel, drag and drop an **Input File** onto the page.
3. Set **value** to be the class created in Step 1.
4. If you want the value of the component to appear as read-only until the user hovers over it, expand the Other 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.

5. From the Component Palette, drag and drop any command component onto the page. This will be used to initiate the upload process.
6. With the command component selected, set the `actionListener` attribute to a listener that will process the file after it has been uploaded.

9.9.2 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 9–12.

Example 9–12 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>
<!-- 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 9–12 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.

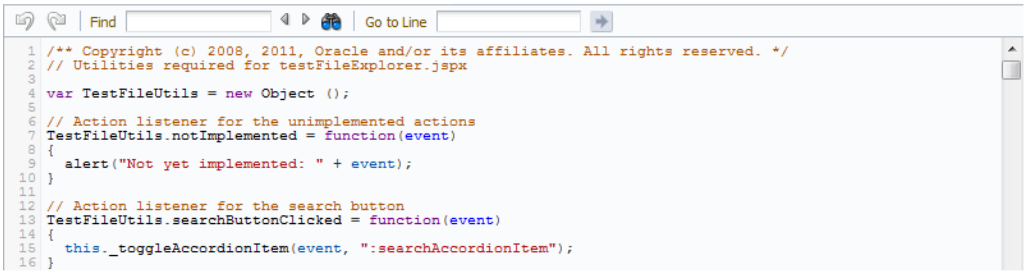
9.10 Using Code Editor

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.

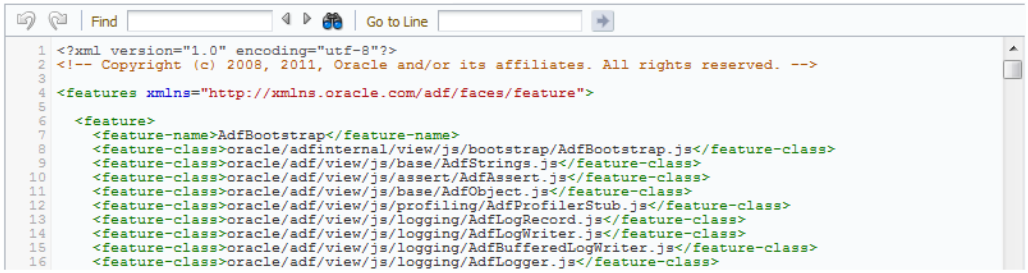
The code editor component supports Javascript, XML, and Groovy languages, as shown in Figure 9–37.

Figure 9–37 Code Editor Component using Javascript, XML, and Groovy

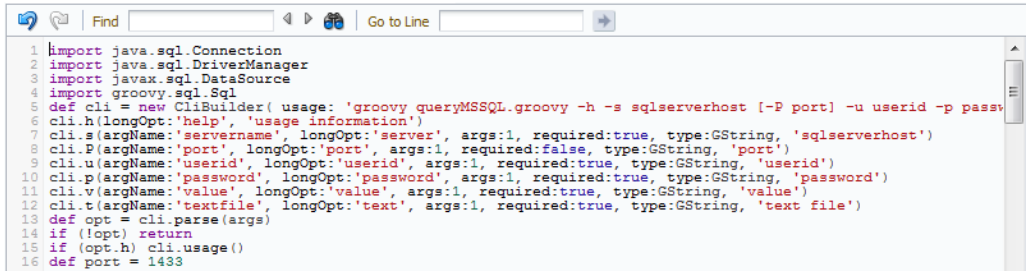
Javascript



XML



Groovy



The code editor component provides the following functionalities:

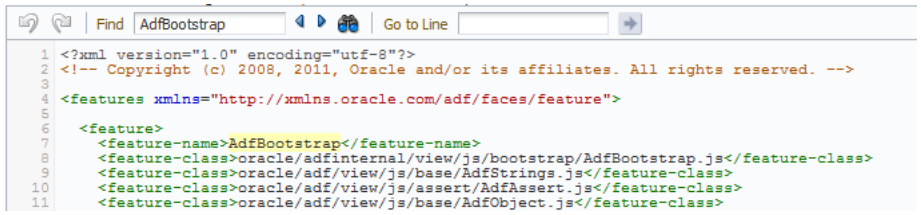
- Line numbering
- Undo and redo operations (also possible using keyboard shortcuts Ctrl+Z and Ctrl+Y)
- Jump to a specific line
- Find and replace
- Color-coded text
- Highlighting syntaxes and search terms
- Auto-indent
- Auto-format
- Message pane for error messages
- Support for large files with more than thousand lines of code

The user can use the toolbar (shown in Figure 9–38) to undo and redo the changes, search and replace text, and jump to a specific line number.

Figure 9–38 Code Editor Toolbar

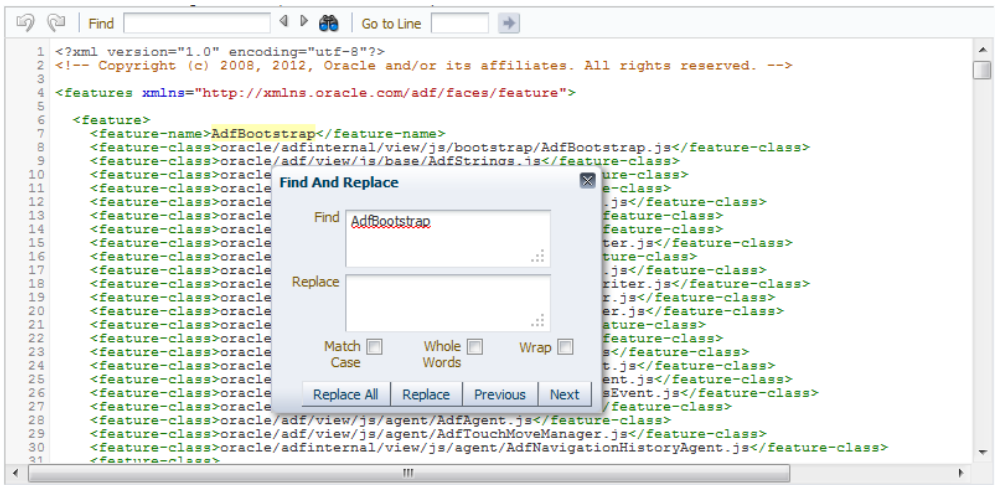
To search for a string, enter the search term in the **Find** field, and click **Find Next** or **Find Previous** icons to locate the search string in the code editor. Figure 9–39 shows the Find field of the toolbar used to search a string in the code editor.

Figure 9–39 Using the Find Field of Code Editor Toolbar



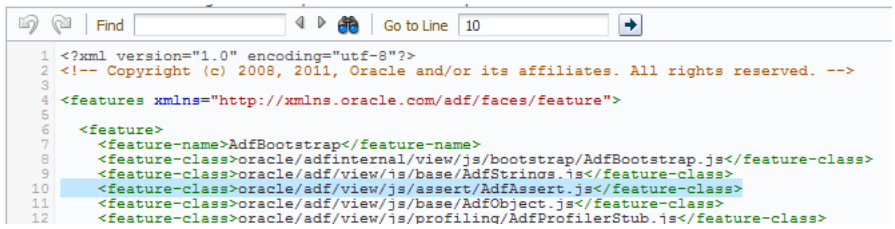
To search a case sensitive string, or replace a search term, open the Find and Replace dialog from the **Find and Replace** icon and perform the operations from the dialog, as shown in Figure 9–40.

Figure 9–40 Using Find and Replace Dialog of Code Editor



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 9–41.

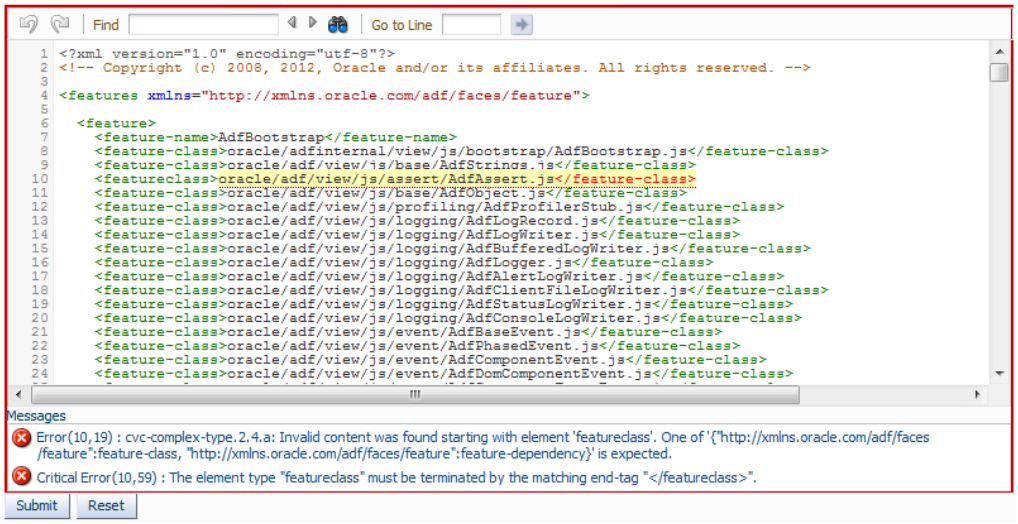
Figure 9–41 Using Go To Line Feature of Code Editor



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 9–42

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

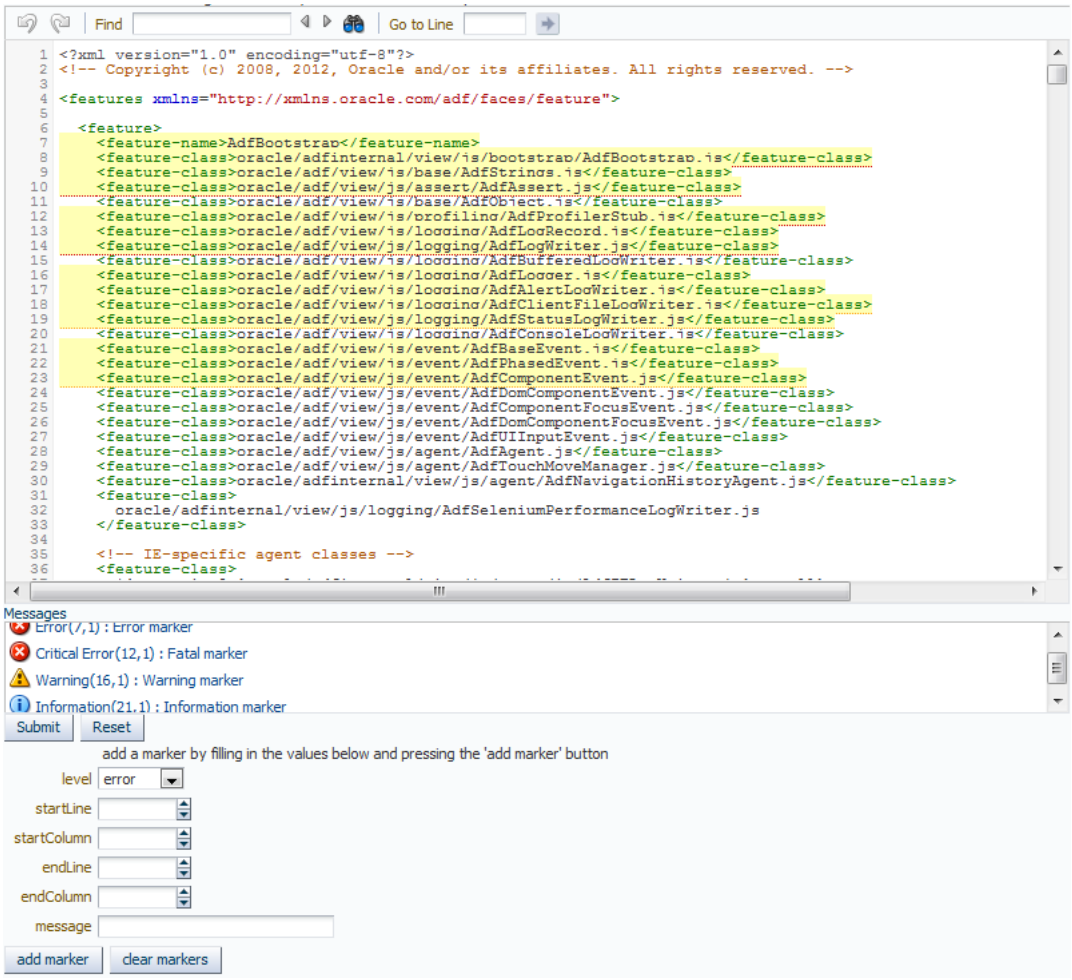
Figure 9–42 Message Pane of Code Editor



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 9–43 shows the code editor with error, critical, warning, and information markers.

Figure 9–43 Using Markers in Code Editor



9.10.1 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.

To add a codeEditor component:

1. In the Component Palette, from the Common Components panel, drag and drop a Code Editor component onto the page.
2. In the Property Inspector, 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.

10

Using Tables and Trees

This chapter describes how to display tables and trees using the ADF Faces `table`, `tree` and `treeTable` components. If your application uses the Fusion technology stack, then you can use data controls to create tables and trees. For more information see the "Creating ADF Databound Tables" and "Displaying Master-Detail Data" chapters of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*

This chapter includes the following sections:

10.1 Introduction to Tables, Trees, and Tree Tables

Structured data can be displayed as tables consisting of rows and 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. Instead of containing a child component for each record to be displayed, and then binding these components to the individual records, table, tree and tree table 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 10.1.5, "Accessing Client Table, Tree, and Tree Table Components."

Example 10–1 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.

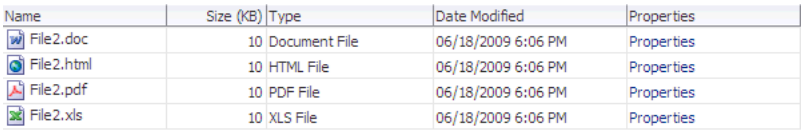
Example 10–1 JSF Code for a Table Uses the var Attribute to Access Values

```
<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>
```

The table component displays simple tabular data. Each row in the table displays one object in a collection, for example one row in a database. The column component displays the value of attributes for each of the objects.

For example, as shown in Figure 10–1, the Table tab in the File Explorer application uses a table to display the contents of the selected directory. The table `value` attribute is bound to the `contentTable` property of the `tableContentView` managed bean in the File Explorer demo.

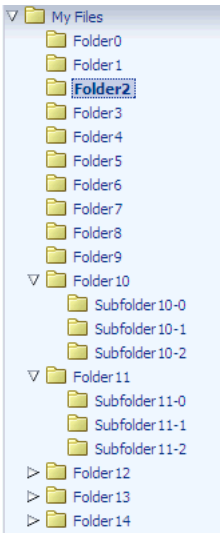
Figure 10–1 Table Component in the File Explorer Application



The table component provides a range of features for end users, such as sorting columns, and selecting one or more rows and then executing an application defined action on the selected rows. It also provides a range of presentation features, such as showing grid lines and banding, row and column headers, column headers spanning groups of columns, and values wrapping within cells.

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 10–2 shows the file directory in the File Explorer application, which is displayed using a tree component.

Figure 10–2 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 10–3.

Figure 10–3 Tree Table in the File Explorer Application

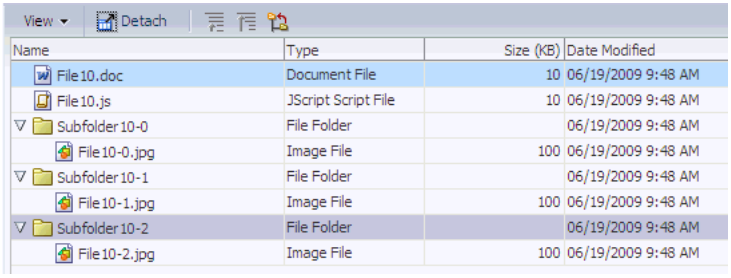
Name	Type	Size (KB)	Date Modified
File11.doc	Document File	10	06/18/2009 6:06 PM
File11.js	JScript Script File	10	06/18/2009 6:06 PM
Subfolder11-0	File Folder		06/18/2009 6:06 PM
File11-0.jpg	Image File	100	06/18/2009 6:06 PM
Subfolder11-1	File Folder		06/18/2009 6:06 PM
File11-1.jpg	Image File	100	06/18/2009 6:06 PM
Subfolder11-2	File Folder		06/18/2009 6:06 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 10–4 shows the tree table from the File Explorer application with the toolbar, menus, and toolbar buttons created using the `panelCollection` component.

Figure 10–4 TreeTable with Panel Collection



10.1.1 Content Delivery

The table, tree, and tree table 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 tables, trees, and tree tables 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 table or tree 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 table 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 table. 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 non-database 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. The `fetchSize` attribute determines the number of rows requested from the client to the server on each attempt to fill the component. 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.

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 table, 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.

10.1.2 Row Selection

You can configure selection to be either for no rows, for a single row, or for multiple rows of tables, trees, and tree tables using the `rowSelection` attribute. 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 command button that navigates to another page where the data for the selected row is displayed and the user can edit it.

When the selected row (or node) of a table, tree, or tree table 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 10.2.8, "What You May Need to Know About Performing an Action on Selected Rows in Tables."

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.

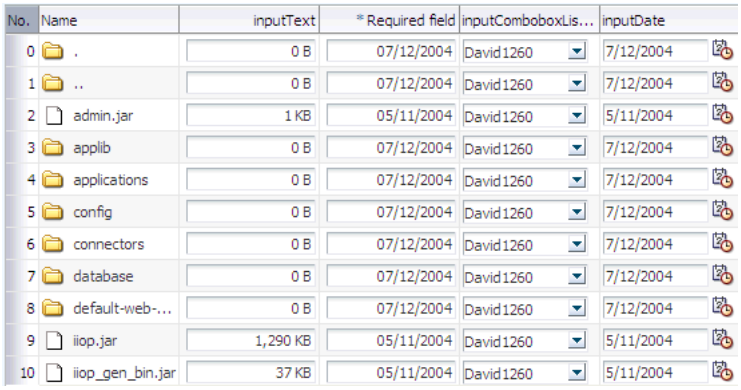
For example, if you configure your table to fetch only 25 rows at a time, but the user selects 100 rows, the framework is able to keep track of the selection.

10.1.3 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 10–5 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).

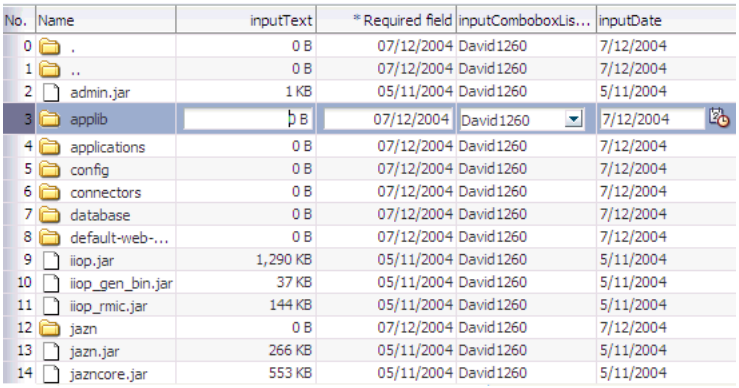
Figure 10–5 Table Whose Rows Can All Be Edited



No.	Name	inputText	* Required field	inputComboboxLis...	inputDate
0	.	0 B	07/12/2004	David1260	7/12/2004
1	..	0 B	07/12/2004	David1260	7/12/2004
2	admin.jar	1 KB	05/11/2004	David1260	5/11/2004
3	applib	0 B	07/12/2004	David1260	7/12/2004
4	applications	0 B	07/12/2004	David1260	7/12/2004
5	config	0 B	07/12/2004	David1260	7/12/2004
6	connectors	0 B	07/12/2004	David1260	7/12/2004
7	database	0 B	07/12/2004	David1260	7/12/2004
8	default-web-...	0 B	07/12/2004	David1260	7/12/2004
9	iiop.jar	1,290 KB	05/11/2004	David1260	5/11/2004
10	iiop_gen_bin.jar	37 KB	05/11/2004	David1260	5/11/2004

Figure 10–6 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 10–5 were used to build the page. The only row that actually renders those components is the active row.

Figure 10–6 Table Allows Only One Row to Be Edited at a Time



No.	Name	inputText	* Required field	inputComboboxLis...	inputDate
0	.	0 B	07/12/2004	David1260	7/12/2004
1	..	0 B	07/12/2004	David1260	7/12/2004
2	admin.jar	1 KB	05/11/2004	David1260	5/11/2004
3	applib	0 B	07/12/2004	David1260	7/12/2004
4	applications	0 B	07/12/2004	David1260	7/12/2004
5	config	0 B	07/12/2004	David1260	7/12/2004
6	connectors	0 B	07/12/2004	David1260	7/12/2004
7	database	0 B	07/12/2004	David1260	7/12/2004
8	default-web-...	0 B	07/12/2004	David1260	7/12/2004
9	iiop.jar	1,290 KB	05/11/2004	David1260	5/11/2004
10	iiop_gen_bin.jar	37 KB	05/11/2004	David1260	5/11/2004
11	iiop_rmic.jar	144 KB	05/11/2004	David1260	5/11/2004
12	jazn	0 B	07/12/2004	David1260	7/12/2004
13	jazn.jar	266 KB	05/11/2004	David1260	5/11/2004
14	jazncore.jar	553 KB	05/11/2004	David1260	5/11/2004

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: You should not use more than one editable component in a column.

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.

10.1.4 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 13.2, "Declaratively Creating Popup Elements."

10.1.5 Accessing Client Table, Tree, and Tree Table Components

With ADF Faces, the contents of the table, tree, or tree table 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 3.1, "Introduction to 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`.

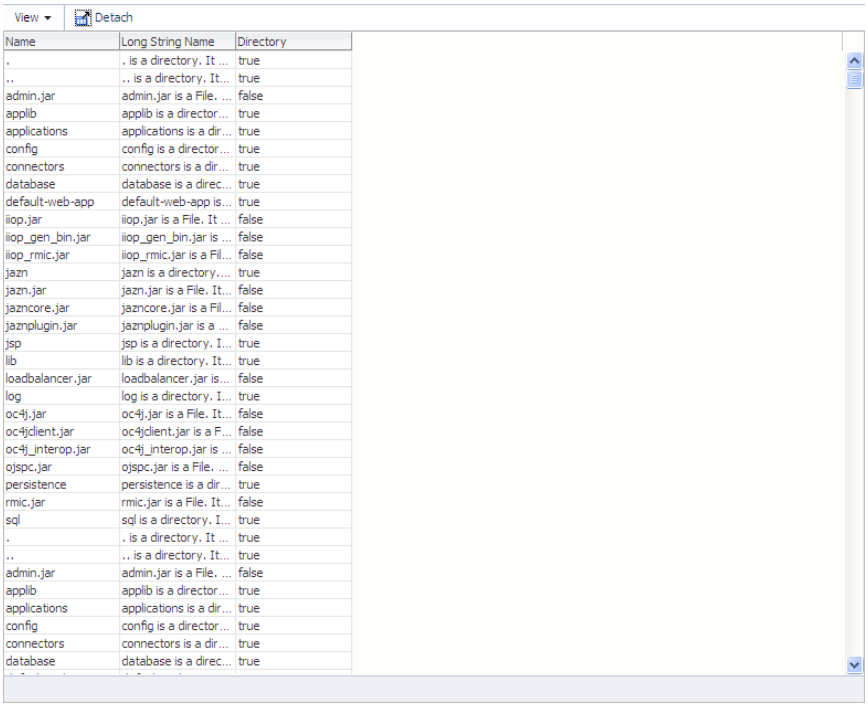
Because there is no client-side support for EL in the rich client framework (RCF), nor is there support for sending entire table models to the client, the client-side code cannot rely on component stamping to access the value. Instead of reusing the same component instance on each row, a new JavaScript 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 10.10, "Accessing Selected Values on the Client from Components That Use Stamping."

10.1.6 Geometry Management and Table, Tree, and Tree Table Components

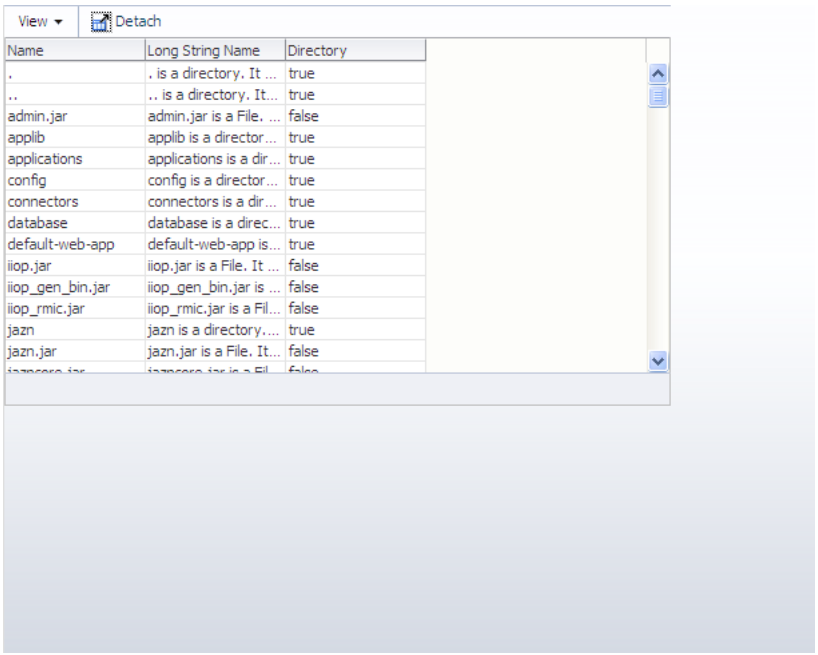
By default, when tables, trees, and tree tables are placed in a component that stretches its children (for example, a `panelCollection` component inside a `panelStretchLayout` component), the table, tree, or tree table will stretch to fill the existing space. However, in order for the columns to stretch 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 10–7.

Figure 10–7 Table Stretches But Columns Do Not



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, as shown in Figure 10–8.

Figure 10–8 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.

10.2 Displaying Data in Tables

The table 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 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 the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

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 10.2.1, "Columns and Column Data."

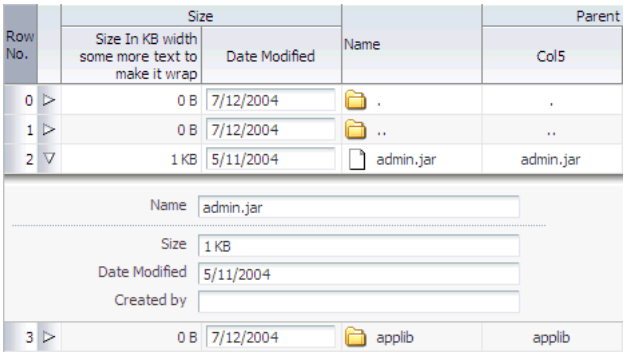
The child components of each column display the data for each row in that column. The column does not create child components per row; instead, the table uses stamping to render each row. Each child is stamped once per row, repeatedly for all the rows. As each row is stamped, the data for the current 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 10.2.9, "What You May Need to Know About Dynamically Determining Values for Selection Components in Tables."

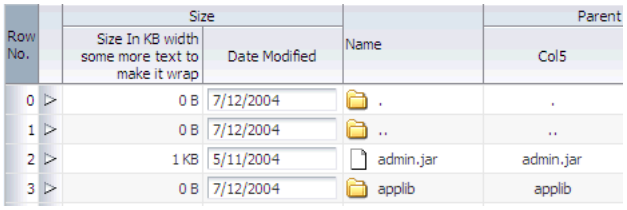
You can use the `detailStamp` facet in a table to include data that can be optionally displayed or hidden. When you add a component to this facet, the table displays an additional column with an expand and collapse icon for each row. When the user clicks the icon to expand, the component added to the facet is displayed, as shown in Figure 10–9.

Figure 10–9 Extra Data Can Be Optionally Displayed



When the user clicks on the expanded icon to collapse it, the component is hidden, as shown in Figure 10–10.

Figure 10–10 Extra Data Can Be Hidden

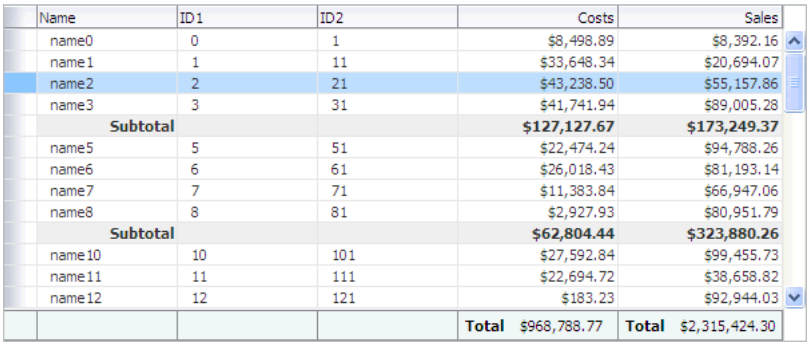


For more information about using the `detailStamp` facet, see Section 10.3, "Adding Hidden Capabilities to a Table."

10.2.1 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 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 10–11 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 10–11 Footer Facets in a Column



Name	ID1	ID2	Costs	Sales
name0	0	1	$8,498.89	$8,392.16
name1	1	11	$33,648.34	$20,694.07
name2	2	21	$43,238.50	$55,157.86
name3	3	31	$41,741.94	$89,005.28
Subtotal			**$127,127.67**	**$173,249.37**
name5	5	51	$22,474.24	$94,788.26
name6	6	61	$26,018.43	$81,193.14
name7	7	71	$11,383.84	$66,947.06
name8	8	81	$2,927.93	$80,951.79
Subtotal			**$62,804.44**	**$323,880.26**
name10	10	101	$27,592.84	$99,455.73
name11	11	111	$22,694.72	$38,658.82
name12	12	121	$183.23	$92,944.03
			Total $968,788.77	**Total** $2,315,424.30

10.2.2 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 10.2.8, "What You May Need to Know About Performing an Action on Selected Rows in Tables."
- 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 10.1.6, "Geometry Management and 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.

- 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 10.1.3, "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 10.8, "Displaying Table Menus, Toolbars, and Status Bars."

10.2.3 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. In order for the table to be able to sort, the underlying data model must also support sorting. For more information, see Section 10.2.7, "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.
- 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 10–12 shows how a table showing departments appears if the first column is configured to be a row header.

Figure 10–12 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 10–13.

Figure 10–13 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.

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 31, "Allowing User Customization on JSF Pages."

10.2.4 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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Once you complete the dialog, and the table and columns are added to the page, you can use the Property Inspector 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.

To display a table on a page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Table** to open the Create ADF Faces Table dialog.

   Use the 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 more information about using the dialog, press F1 or click **Help**.

2. In the Property Inspector, 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 10.2.8, "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`.

3. 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 Property Inspector, and then manually change it:

   1. In the Structure window, right-click the component created by the dialog.
   2. Choose **Convert** from the context menu.
   3. Select the desired component from the list. You can then use the Property Inspector 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`.

4. 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 percentage or as a number of pixels. 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 8.2.1, "Geometry Management and Component Stretching."

   - **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 (non-stretched) 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 user can change the values of the column width at runtime, 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 31, "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 10.4, "Enabling Filtering in Tables."

- 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.

5. Expand the Behavior section. You use this section to configure the behavior of the table by setting the following table-specific attributes:

   - **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 10.8, "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 31, "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 10.1.1, "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 10.1.1, "Content Delivery."

   - **AutoHeightRows**: If you want your table to size the height automatically to fill up available space, specify the maximum number of rows that the table should display. The default value is -1 (no automatic sizing for any number of rows). You can also set the value to 0 to have the value be the same as the `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). For more information, see Section 8.2.1, "Geometry Management and Component Stretching."

- **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.

- **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.

- **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 10.1.3, "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, located in the Other section.

- **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 13, "Using Popup Dialogs, Menus, and Windows."
- **FilterModel**: Use in conjunction with `filterVisible`. For more information, see Section 10.4, "Enabling Filtering in Tables."
- Various listeners: Bind listeners to methods that will execute when the table invokes the corresponding event (the `columnSelectionListener` is

located in the Other section). For more information, see Chapter 5, "Handling Events."

6. Expand the Other section, and set the following:

   - **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.
   - **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, 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 31, "Allowing User Customization on JSF Pages."

7. In the Structure window, select a column. In the Property Inspector, 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 12). 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 20, "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 10.2.7, "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.

   - **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 10.4, "Enabling Filtering in Tables."

---

8. 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.
   - **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.

9. 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 they 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.

     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.

10. To add a column to an existing table, in the Structure window, right-click the table and from the context menu choose **Insert Inside Table** > **Column**.
11. To add facets to the table, right-click the table and from the context menu, choose **Facets - Table** and choose the type of facet you want to add. You can then add a component directly to the facet.

    Tip: Facets can have only one direct child. If you want the facet to display more than one component, first insert a group component (such as `panelGroupLayout`) and then insert the multiple components as children to the group component.

12. To add facets to a column, right-click the column and from the context menu, choose **Facets - Column** and choose the type of facet you want to add. You can then add a component directly to the facet.

    Tip: Facets can have only one direct child. If you want the facet to display more than one component, first insert a group component (such as `panelGroupLayout`) and then insert the multiple components as children to the group component.

13. 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"`.

10.2.5 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 Property Inspector.

Example 10–2 shows abbreviated page code for the table in the File Explorer application.

Example 10–2 ADF Faces Table in the File Explorer Application

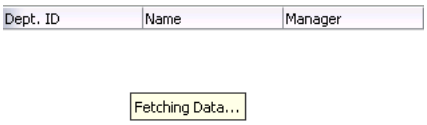
```
<af:table id="folderTable" var="file"
          value="#{explorer.contentViewManager.
                                  tableContentView.contentModel}"
```

```
            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" 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"
             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>
```

10.2.6 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 10–14 shows a page containing a table during the second PPR request.

Figure 10–14 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. The table responds to this event by calling the `setSortCriteria()` method on the underlying `CollectionModel` instance, and calls any registered `SortListener` instances.

10.2.7 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)
```

For more information, see the MyFaces Trinidad Javadoc at `http://myfaces.apache.org/trinidad/trinidad-1_2/trinidad-api/apidocs/index.html`.

If the underlying model is not a `CollectionModel` instance, the `table` component automatically examines the actual data to determine which properties can be sorted. Any column that has data that implements the `java.lang.Comparable` class is able to be sorted. Although this automatic support is not as efficient as coding sorting directly into a `CollectionModel` (for instance, by translating the sort into an `ORDER BY` SQL clause), it may be sufficient for small data sets.

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

10.2.8 What You May Need to Know About Performing an Action on Selected Rows in Tables

A table can allow users to select one or more rows and perform some actions on those rows.

When the selection state of a table changes, the table 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 table, you can register a listener on the table either using the `selectionListener` attribute or by adding a listener to the table 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 table. 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 table. 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 10–3 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 10–3 Selecting Rows

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

Example 10–4 shows an actions method, performDelete, which iterates through all the selected rows and calls the markForDeletion method on each one.

Example 10–4 Using the rowKey Object

```
public void performDelete(ActionEvent action)
{
  UIXTable table = getTable();
  Iterator selection = table.getSelectedRowKeys().iterator();
  Object oldKey = table.getRowKey();
  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();
  }
  // restore the old key:
  table.setRowKey(oldKey);
}

// Binding methods for access to the table.
public void setTable(UIXTable table) { _table = table; }
public UIXTable getTable() { return _table; }
private UIXTable _table;
```

10.2.9 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 10–5.

Example 10–5 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 10–6.

Example 10–6 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>
```

10.2.10 What You May Need to Know About Using the Iterator Tag

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 Example 10–7.

Example 10–7 Using the Iterator Tag

```
<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:inputText label="Element Name" value="#{row.name}"/>
  <af:inputText label="Atomic Number" value="#{row.number}"/>
  <af:inputText label="Symbol" value="#{row.symbol}"/>
  <af:inputText 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.

10.3 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 10–15 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 22, "Developing Accessible ADF Faces Pages."

Figure 10–15 Table with Unexpanded DetailStamp Facet

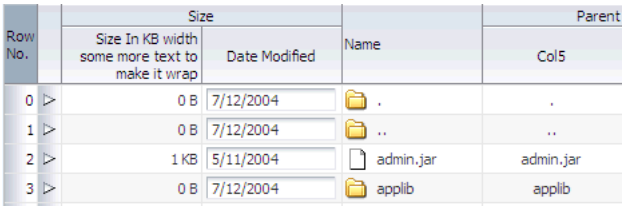
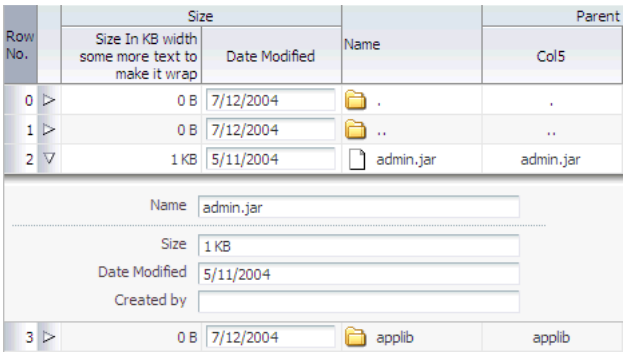


Figure 10–16 shows the same table, but with the `detailStamp` facet expanded for the first row.

Figure 10–16 Expanded detailStamp Facet



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.

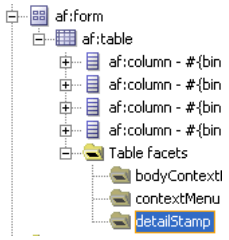
10.3.1 How to Use the detailStamp Facet

To use the `detailStamp` facet, you insert a component that is bound to the data to be displayed or hidden into the facet.

To use the detailStamp facet:

1. In the Component Palette, drag the components you want to appear in the facet to the **detailStamp** facet folder. Figure 10–17 shows the **detailStamp** facet folder in the Structure window.

Figure 10–17 detailStamp Facet in the Structure Window



Tip: If the facet folder 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 10–8 shows abbreviated code used to display the `detailStamp` facet shown in Figure 10–16, which shows details about the selected row.

Example 10–8 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>
```

> **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.

10.3.2 What Happens at Runtime: Disclosing Row Data

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.

10.4 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 10–18 has been filtered to display only rows in which the `Location` value is 1700.

Figure 10–18 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 the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*

In Example 10–9, 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 10–9 Table Component with Filtering Enabled

```
<af:table value="#{myBean.products}" var="row"
          ...
          filterVisible="true"
          ...
          rowselection="single">
          ...
  <af:column sortProperty="ProductID" filterable="true" sortable="true"
    <af:outputText value="#{row.ProductID}">
          ...
  </af:column>
  <af:column sortProperty="Name" filterable="true" sortable="true"
    <af:outputText value="#{row.Name}"/>
          ...
  </af:column>
  <af:column sortProperty="warehouse" filterable="true" sortable="true"
    <af:outputText value="#{row.warehouse}"/>
          ...
  </af:column>
</af:table>
```

10.4.1 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 table that will use filtering must either have a value for its `headerText` attribute, or it must contain a component in the `header` facet of the column that is to be filtered. This allows the filter component to be displayed. Additionally, the column must be configured to be sortable, because the `filterModel` class uses the `sortProperty` attribute.

To add filtering to a table:

1. Create a Java class that is a subclass of the `FilterableQueryDescriptor` class. For more information about this class, see the `ADF Faces Javadoc`.
2. Create a table, as described in Section 10.2, "Displaying Data in Tables."
3. Select the table in the Structure window and set the following attributes in the Property Inspector:

- **FilterVisible**: Set to `true` to display the filter criteria input field above searchable column.
- **FilterModel**: Bind to an instance of the `FilterableQueryDescriptor` class created in Step 1.

  Tip: If you want to use a component other than an `inputText` component for your filter (for example, an `inputDate` component), then instead of setting `filterVisible` to `true`, you can add the needed component to the `filter` facet. To do so:

  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 Component Palette, drag and drop a component into the facet.
  3. Set the value of the component to the corresponding attribute within the `FilterableQueryDescriptor` class created in Step 1. Note that the value must take into account the variable used for the row, for example:

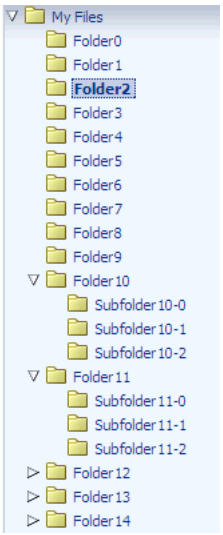
     ```
     #{af:inputDate label="Select Date" id="name"
                         value="row.filterCriteria.date"}
     ```

4. In the Structure window, select a column in the table and in the Property Inspector, 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.

10.5 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. As an example, in an organization chart, each element, that is, each employee, in the hierarchy may have any number of child elements (direct reports). 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 10–19 shows a tree used to display directories in the File Explorer application.

Figure 10–19 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 10.2, "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 `oracle.adf.view.rich.model.ChildPropertyTreeModel` class useful when constructing a `TreeModel` class, as shown in Example 10–10.

Example 10–10 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 the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*

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 10.5.4, "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** context menu item (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.

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 10–11, 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 10–11 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 10.8, "Displaying Table Menus, Toolbars, and Status Bars."

10.5.1 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.

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 10–10.
2. In the Component Palette, from the Common Components panel, drag and drop a **Tree** to open the Insert Tree dialog. Configure the tree as needed. Click **Help** or press F1 for help in using the dialog.
3. In the Property Inspector, 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

4. 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.
5. 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 10.1.3, "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 13, "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 10.5.5, "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 10.1.1, "Content Delivery."
   - **FetchSize**: Specify the number of rows in the data fetch block. For more information, see Section 10.1.1, "Content Delivery."
   - **SelectionListener**: Optionally enter an EL expression for a listener that handles selection events. For more information, see Section 10.5.5, "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.
6. 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 10.5.4, "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 10.5.5, "What You May Need to Know About Programmatically Selecting Nodes."

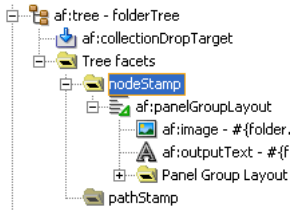
7. If you want your tree to size its height automatically, expand the Other section and set **AutoHeightRows** to the maximum number of nodes to display before a scroll bar is displayed. The default value is -1 (no automatic sizing for any number of number). You can set the value to 0 to have the value 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 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). For more information, see Section 10.1.6, "Geometry Management and Table, Tree, and Tree Table Components."

8. To add components to display data in the tree, drag the desired component from the Component Palette to the **nodeStamp** facet. Figure 10–20 shows the **nodeStamp** facet for the tree used to display directories in the File Explorer application.

Figure 10–20 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 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 10–20.

10.5.2 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 10–12 shows the abbreviated code for the tree in the File Explorer application that displays the directory structure.

Example 10–12 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>
```

10.5.3 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 10.5.4, "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 10.5.5, "What You May Need to Know About Programmatically Selecting Nodes."

10.5.4 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 10–13.

Example 10–13 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 10–14. The example illustrates expansion of a tree node. For the contraction of a tree node, you would use `getRemovedSet`.

Example 10–14 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 10–15. 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 10–15 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 10–16.

Example 10–16 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;
}
```

10.5.5 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.

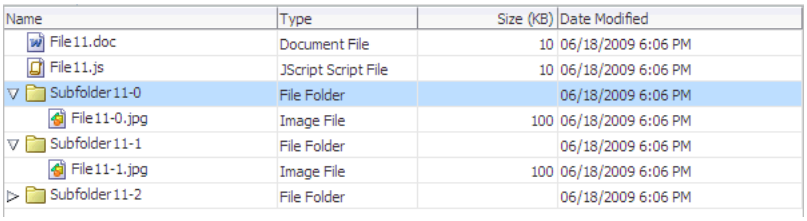
When a user selects or deselects a node, the tree component fires a `selectionEvent` event. This event has two `RowKeySet` objects: the `RemovedSet` object for all the deselected nodes and the `AddedSet` object for all the selected nodes.

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 10.2.8, "What You May Need to Know About Performing an Action on Selected Rows in Tables."

10.6 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 10–21 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 10–21 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 10.5, "Displaying Data in Trees").

For example, in the File Explorer application (as shown in Figure 10–21), 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 10–17 shows abbreviated code for the tree table in the File Explorer application.

Example 10–17 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 10.2, "Displaying Data in Tables" and Section 10.5, "Displaying Data in Trees."

10.6.1 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 Property Inspector to configure additional attributes on the tree table.

To add a tree table to a page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Tree Table** onto the page to open the Insert Tree Table wizard. Configure the table by completing the wizard. If you need help, press F1 or click **Help**.
2. Use the Property Inspector 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 10.2.4, "How to Display a Table on a Page," and Section 10.5.1, "How to Display Data in Trees" for help in configuring the attributes.

10.7 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 4.7, "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 command button to navigate to a new page to edit the data for that row, as shown in

Example 10–18. The EL variable name `emp` is used to represent one row (employee) in the table. The `action` attribute value of the `commandButton` 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 10–18 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:commandButton text="Show more details" action="showEmpDetail">
      <af:setPropertyListener from="#{emp}"
                              to="#{pageFlowScope.empDetail}"
                              type="action"/>
    </af:commandButton>
  </af:column>
</af:table>
```

When the user clicks the command 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 10–19.

Example 10–19 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>
```

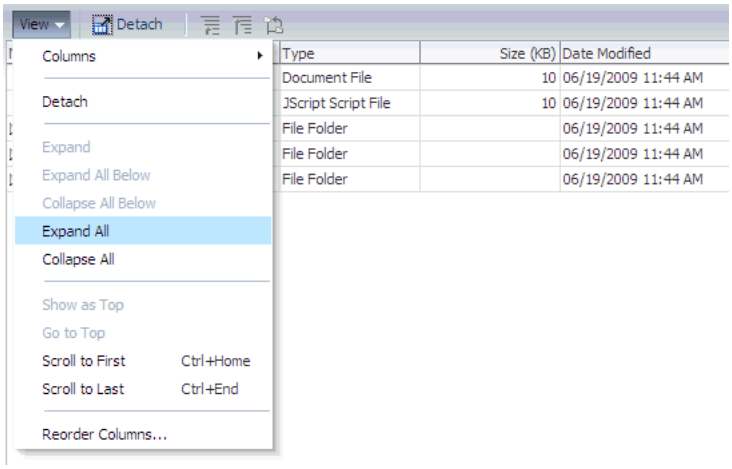
10.8 Displaying Table Menus, Toolbars, and Status Bars

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 10–22 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 14, "Using Menus, Toolbars, and Toolboxes."

Figure 10–22 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 10–20 shows how the `panelCollection` component contains menus and toolbars.

Example 10–20 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:commandToolbarButton shortDesc="Create" icon="/new_ena.png">
    </af:commandToolbarButton>
    <af:commandToolbarButton shortDesc="Update" icon="/update_ena.png">
    </af:commandToolbarButton>
    <af:commandToolbarButton shortDesc="Delete" icon="/delete_ena.png">
    </af:commandToolbarButton>
  </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 14.2, "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.

10.8.1 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.

To create a panelCollection component with an aggregate display component:

1. In the Component Palette, 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 Property Inspector, set the `featuresOff` attribute. Table 10–1 shows the valid values and the corresponding effect on the toolbar.

Table 10–1 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 many `commandMenuItem` components.

From the Component Palette, 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 14, "Using Menus, Toolbars, and Toolboxes."

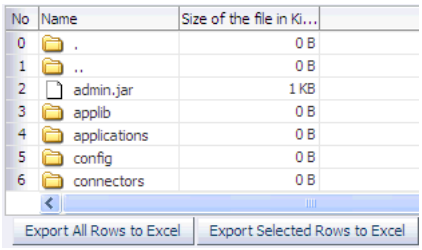
10.9 Exporting Data from Table, Tree, or Tree Table

You can export the data from a table, tree, or tree table, or from a table region of the DVT project Gantt chart to a Microsoft Excel spreadsheet. To allow users to export a table, you create an action source, such as a command button or command link, and add an `exportCollectionActionListener` component and associate it with the data you wish to export. You can configure the table so that all the rows 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 26.8, "Exporting from a Pivot Table."

For example, Figure 10–23 shows the table from the ADF Faces demo that includes a command button component that allows users to export the data to an Excel spreadsheet.

Figure 10–23 Table with Command Button for Exporting Data



When the user clicks the command button, the listener processes the exporting of all the rows to Excel. As shown in Figure 10–23, you can also configure the `exportCollectionActionListener` component so that only the rows the user selects are exported.

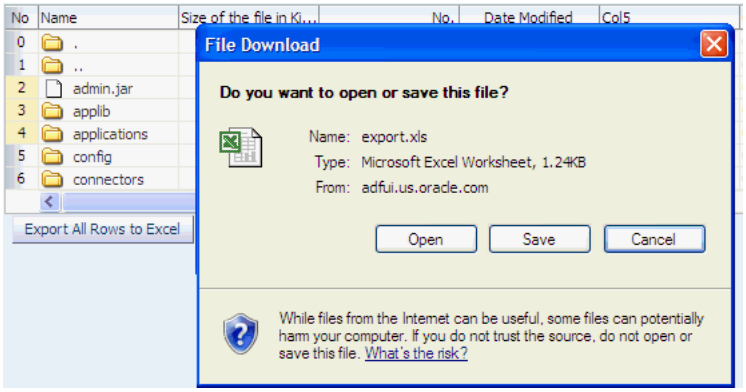
Note: Only the following can be exported:

- 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.

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 spreadsheet as shown in Figure 10–24, or the spreadsheet 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 10–24 Exporting to Excel Dialog



If the user chooses to save the file, it can later be opened in Excel, as shown in Figure 10–25. 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, the spreadsheet opens in the browser window. If the user is viewing the page in Mozilla Firefox, the spreadsheet opens in Excel.

Figure 10–25 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: 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.

10.9.1 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:

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 10.2, "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 10.2.2, "Formatting Tables."

To export collection data to an external format:

1. In the Component Palette, from the Common Components 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 10.8, "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 Excel**.

2. In the Component Palette, 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`.

4. With the `exportCollectionActionListener` component still selected, in the Property Inspector, 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 Excel.
   - **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 10.3, "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.

Example 10–21 shows the code for a table and its `exportCollectionActionListener` component. Note that the `exportedId` value is set to the table `id` value.

Example 10–21 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:commandButton text="Export To Excel" immediate="true">
  <af:exportCollectionActionListener type="excelHTML" exportedId="table"
                                filename="export.xls" title="ADF Faces Export"/>
```

10.9.2 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.

10.10 Accessing Selected Values on the Client from Components That Use Stamping

Since there is no client-side support for EL in the rich client framework, nor is there support for sending entire table 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 stamped component itself to access the value. To do this without a client-side data model, you use a client-side selection change listener.

10.10.1 How to Access Values from a Selection in Stamped Components.

To access values on the client from a stamped 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 stamped component to determine the row-specific name, and finally interact with the selected data as needed.

To access selected values from stamped components:

1. In the Structure window for your page, select the component associated with the stamped row. For example, in Example 10–22 the table uses an `outputText` component to display the stamped rows.

Example 10–22 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 Property Inspector 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 Component Palette, from the Operations panel, drag and drop a **Client Listener** as a child to the table.
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 10–23.

Example 10–23 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 10–24 extracts the first row key (which in this case, is the only row key).

Example 10–24 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 table from the source of the event and then use the table to find the instance using the ID and row key. Example 10–25 shows this, where `nameStamp` is the ID of the table.

Example 10–25 Finding a Stamped Component Instance Given a Selected Row

```
// We need the table to find our 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 10–26 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 10–22)and then display the name in an alert.

Example 10–26 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 10–27 shows the entire code for the JavaScript.

Example 10–27 JavaScript Used to Access Selected Row Value

```
function showSelectedName(event)
{
  var firstRowKey;
  var addedRowKeys = event.getAddedSet();

  for (var rowKey in addedRowKeys)
  {
    firstRowKey = rowKey;
    break;
  }
  // We need the table to find our 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);
  }
}
```

10.10.2 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.

11

Using List-of-Values Components

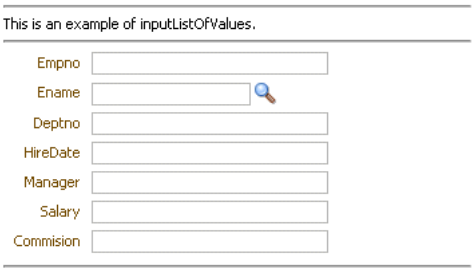
This chapter describes how to use a list-of-values component to display a model-driven list of objects from which a user can select a value.

This chapter includes the following sections:

11.1 Introduction to List-of-Values Components

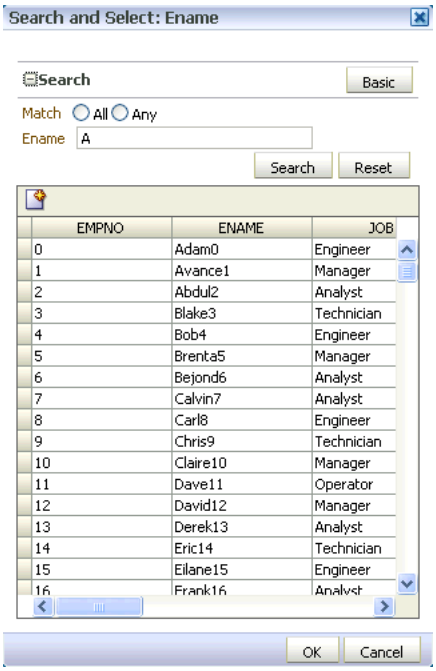
ADF Faces provides two list-of-values (LOV) input components that can display multiple attributes of each list item and can optionally allow the user to search for the needed item. These LOV components are useful when a field used to populate an attribute for one object might actually be contained in 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 11–1.

Figure 11–1 List-of-Values Input Field



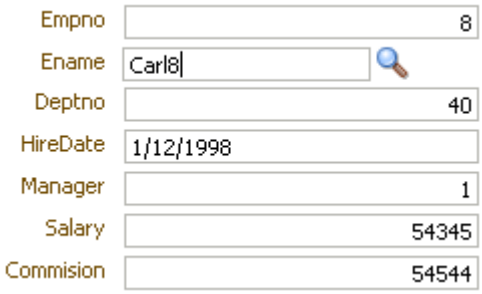
In this form, the employee name field is an LOV that contains a list of employees. When the user clicks the search icon of the `inputListOfValues` component, a Search and Select popup dialog displays all employees, along with a search field that allows the user to search for the employee, as shown in Figure 11–2.

Figure 11–2 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 11–3. The user can then edit and save the data.

Figure 11–3 Form Populated Using LOV Component



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. This chapter describes only the `inputListOfValues` and `inputComboboxListOfValues` LOV components. For more information about select choice components, list box components, and radio buttons, see Chapter 9, "Using Input Components and Defining Forms."

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. Note that the columns of the table will not stretch to the full width of the dialog.

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 ReturnPopupEvent to the `returnPopupListener`. When you implement the `returnPopupListener`, you can perform functions such as setting the value of the LOV component, its dependent components, and displaying the custom content. In the `searchContent` facet you can add components such as tables, trees, and inputText 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 11–1 show the code to display custom content using a table component.

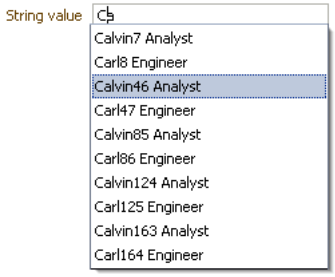
Example 11–1 Adding Custom Content to the Search and Select Dialog

```
<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>
```

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.

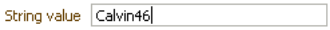
You can also implement the LOV component to automatically display a list of suggested items when the user types in a partial value. For example, when the user enters `Ca`, then a suggested list which partially matches `Ca` is displayed as a suggested items list, as shown in Figure 11–4.

Figure 11–4 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 11–5.

Figure 11–5 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. You also need to set the component's `autoSubmit` property to `true`.

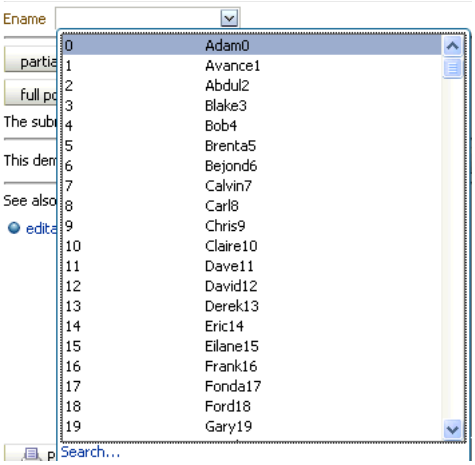
In your LOV model implementation, you can implement a *smart list* that filters the list further. You can implement a smart list for both LOV components. If you are using ADF Model, the `inputComboboxListOfValues` allows you declaratively select a smart list filter defined as a view criteria for that 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 11–2 shows the code for an LOV component with both auto suggest behavior and smart list.

Example 11–2 Auto Suggest Behavior and Smart List

```
<af:autoSuggestBehavior
    suggestItems="#{bean.suggestItems}"
    smartList="#{bean.smartList}"
    maxSuggestedItems="7"/>
```

Figure 11–6 shows how a list can be displayed by an `inputComboboxListOfValues` component. If the popup dialog includes a query panel, then a **Search** link is displayed at the bottom of the dropdown list. If a query panel is not used, a **More** link is displayed.

Figure 11–6 InputComboboxListOfValues Displays a List of Employee Names



The dropdown list of the `inputComboboxListOfValues` component can display the following:

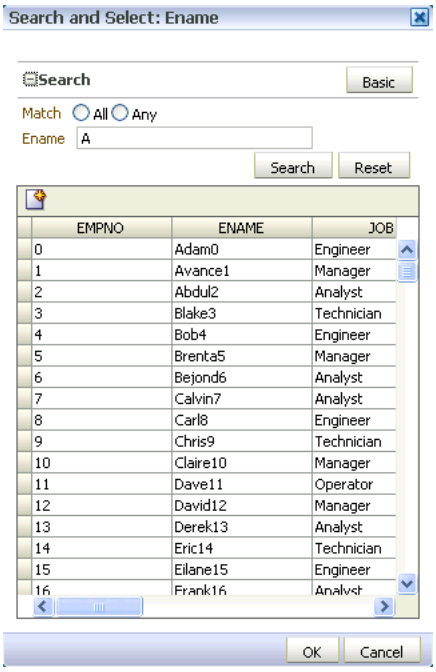
- Full list: As shown in Figure 11–6, a complete list of items returned by the `ListOfValuesModel.getItems()` method.
- Favorites list: A list of recently selected items returned by the `ListOfValuesModel.getRecentItems()` method.
- Search link: A link that opens a popup Search and Select dialog. The 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 `commandLink` components. You are responsible for implementing any logic for the `commandLink` 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 `commandToolbarButton` is displayed with a create icon. At runtime, a `commandToolbarButton` component appears in the LOV popup dialog, as shown in Figure 11–7.

Figure 11–7 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 `commandLink` 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.

> **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 of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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`.

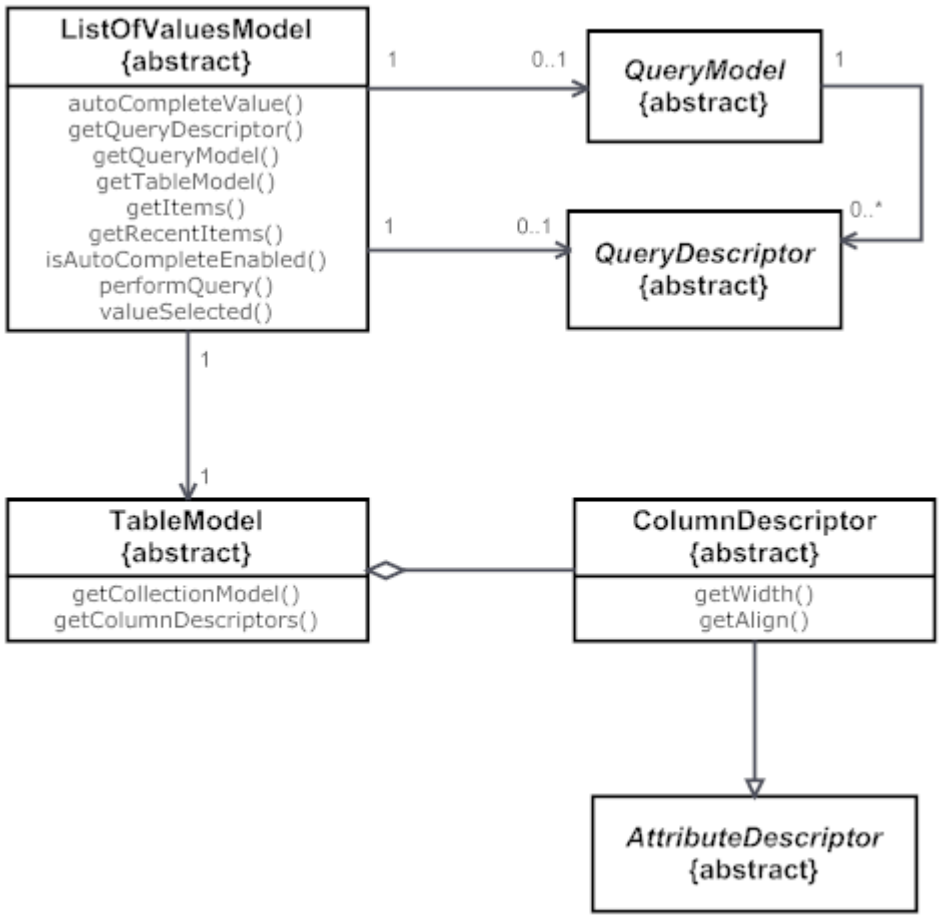
When the user clicks the **Search** button to start a search, the `ListOfValuesModel.performQuery()` method is invoked and the search is performed. For more information about the query model, see Chapter 12, "Using Query Components."

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.

11.2 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. Figure 11–8 shows the class diagram for a `ListOfValues` model.

Figure 11–8 Class Diagram for LIstOfValues Model



To create a ListOfValues model and associated events:

1. Create implementations of each of the interface classes shown in Figure 11–8. Table 11–1 provides a description of the APIs.

Table 11–1 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.
`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.

Table 11–1 (Cont.) ListOfValues Model API

Method	Functionality
`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.

11.3 Using the inputListOfValues Component

The `inputListOfValues` component uses the `ListOfValues` model you implemented to access the list of items, as documented in Section 11.2, "Creating the ListOfValues Data Model."

Before you begin:

You should already have a created 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 Component Palette, from the Common panel, drag an **Input List Of Values** component and drop it onto the page.
2. In the Property Inspector, expand the **Common** section and set the following attributes:
   - **model**: Enter an EL expression that resolves to your `ListOfValuesModel` implementation, as created in Section 11.2, "Creating the ListOfValues Data Model."

- **value**: Enter an EL expression that resolves to the attribute values used to populate the list, as created in Section 11.2, "Creating 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 any other input component. For more information, see Section 9.3, "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. If you are adding the auto suggest behavior, you must set `autoSubmit` to `true`.
   - **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 `commandToolbarButton` 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 any other input component. For more information, see Section 9.3, "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 13, "Using Popup Dialogs, Menus, and Windows."
6. In the Component Palette, from the Operations panel, drag an **Auto Suggest Behavior** and drop it as a child to the `inputListOfValues` component.
7. In the Property Inspector, 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 11–3

Example 11–3 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 11–4.

Example 11–4 suggestItems Method Signature

```
List<javax.model.SelectItem> suggestItems(javax.faces.context.FacesContext,
    oracle.adf.view.rich.model.AutoSuggestUIHints)
```

11.4 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 11.2, "Creating the ListOfValues Data Model."

Before you begin:

You should already have a created a page or page fragment.

To add an inputComboboxListOfValues component:

1. In the Component Palette, from the Common panel, drag an **Input Combobox List Of Values** and drop it onto the page.
2. In the Property Inspector, expand the **Common** section and set the following attributes:

- **model**: Enter an EL expression that resolves to your `ListOfValuesModel` implementation, as created in Section 11.2, "Creating the ListOfValues Data Model."
- **value**: Enter an EL expression that resolves to the attribute values used to populate the list, as created in Section 11.2, "Creating 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 any other input component. For more information, see Section 9.3, "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. If you are adding the auto-suggest behavior, you must set `autoSubmit` to `true`.
   - **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 `commandToolbarButton` 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 any other input component. For more information, see Section 9.3, "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 13, "Using Popup Dialogs, Menus, and Windows."
7. In the Component Palette, from the Operations panel, drag an **Auto Suggest Behavior** and drop it as child to the `inputComboboxListOfValues` component.
8. In the Property Inspector, 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 11–5.

Example 11–5 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 11–6.

Example 11–6 suggestItems Method Signature

```
List<javax.model.SelectItem> suggestItems(javax.faces.context.FacesContext,
    oracle.adf.view.rich.model.AutoSuggestUIHints)
```

12 Using Query Components

This chapter describes how to use the `query` and `quickQuery` search panel components.

This chapter includes the following sections:

12.1 Introduction to 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:

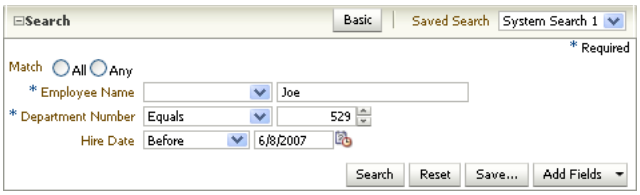
- Selecting and searching against multiple search criteria
- Dynamically adding and deleting criteria items
- Selecting search operators (associated to a single criterion)
- Choosing match all or match any conjunction
- Displaying in a basic, advanced, compact, simple, or design mode
- Creating saved 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 12–1 shows an advanced mode `query` component with three search criteria.

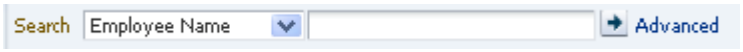
Figure 12–1 Query Component with Three Search Criteria



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 12–1 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 31, "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 12–2 shows a `quickQuery` component in horizontal layout.

Figure 12–2 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 ADF Faces Javadoc.

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 ADF Faces Javadoc. 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 12–2.

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.

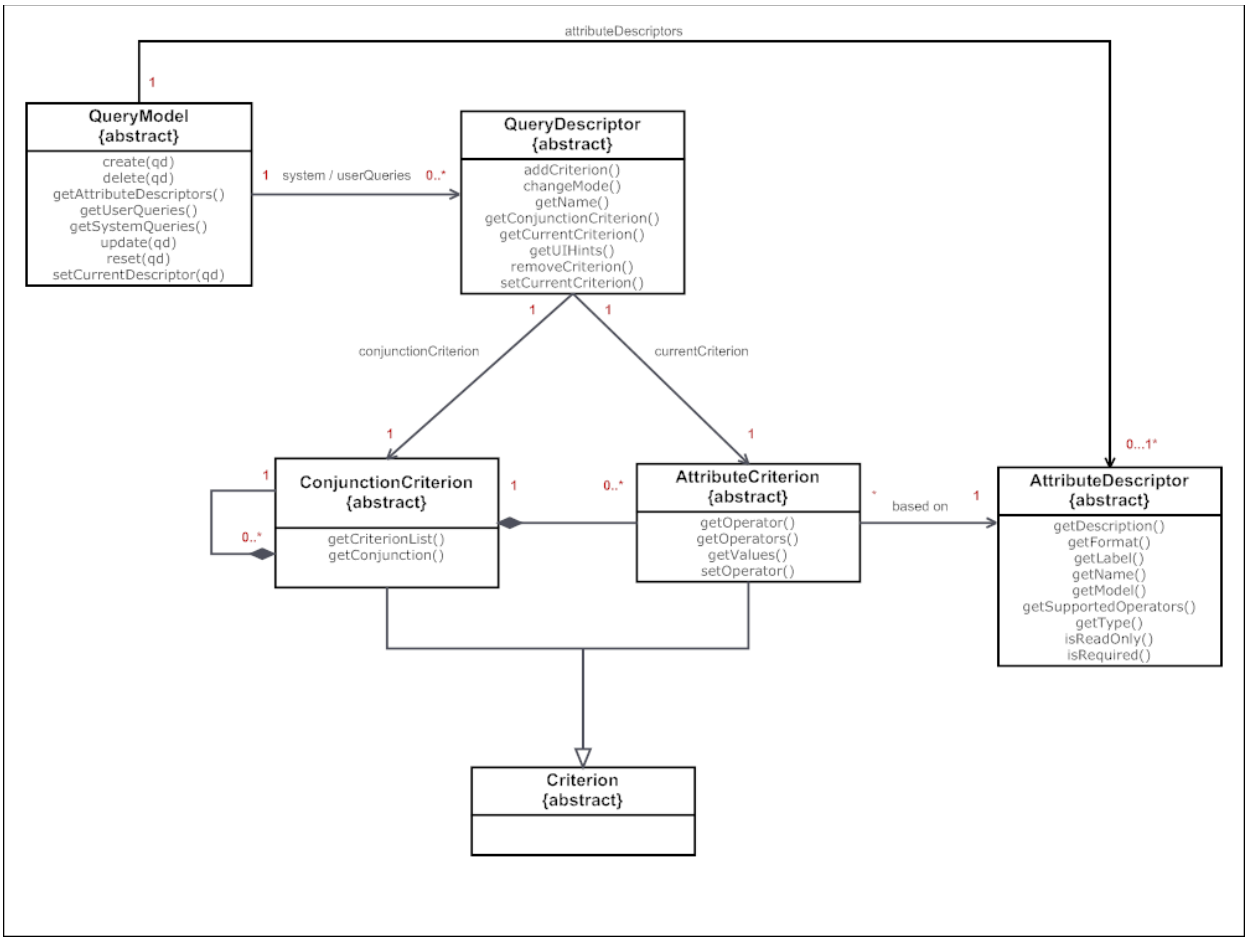
12.2 Implementing the Model for Your Query

Before you can use the `query` components, you must to create your `QueryModel` classes.

> **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 12.3.2, "How to Use a quickQuery Component Without a Model."

Figure 12–3 shows the class diagram for a `QueryModel` class.

Figure 12–3 Class Diagram for QueryModel



To create the model classes:

1. Create implementations of each of the interface classes shown in Figure 12–3. 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 12–1 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 12–1 A QueryListener Handler Method

```
public void processQuery(QueryEvent event)
{
  DemoQueryDescriptor descriptor = (DemoQueryDescriptor) event.getDescriptor();
  String sqlString = descriptor.getSavedSearchDef().toString();
  setSqlString(sqlString);
}
```

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 12–1 and Table 12–2 map the functionality found in the UI component shown in Figure 12–4 with the corresponding interface.

Figure 12–4 Query Component and Associated Popup Dialog

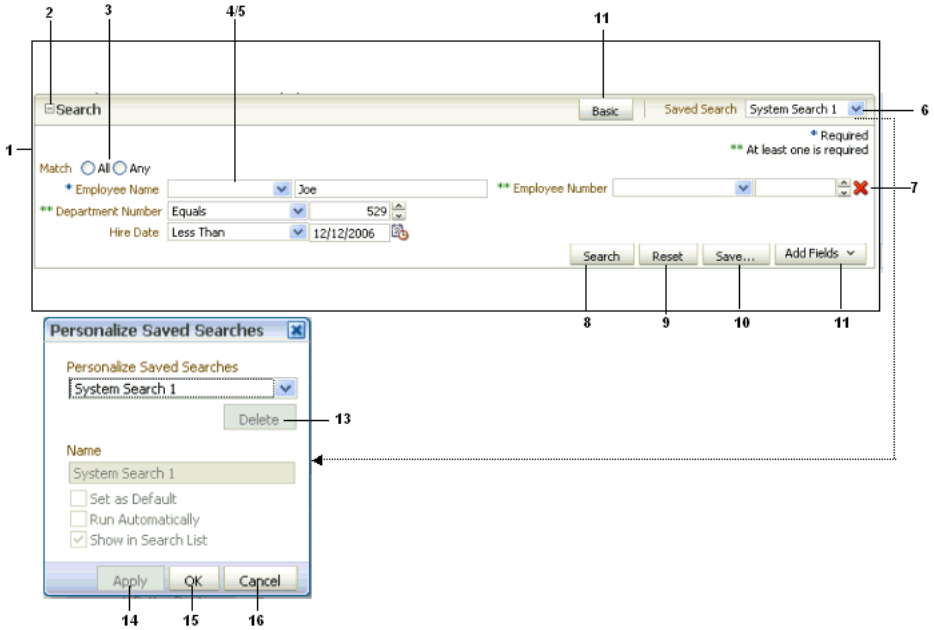


Table 12–1 shows UI artifacts rendered for the `query` component, the associated class, class property, and methods used by the artifact.

Table 12–1 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 12–1 (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. 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 12–2 shows the behaviors of the different UI artifacts, and the associated methods invoked to execute the behavior.

Table 12–2 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 12.4.1, "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 12–2 (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 12–2 (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	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 `QueryOperation Listener` is registered with the `query` component that in turn calls the model methods).	Applies changes made to the selected saved search.
15	OK button	Same as the Apply button.	`QueryOperationEvent` (an internal `QueryOperation Listener` 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.
16	Cancel button	No method defined for this action.	`QueryOperationEvent` (an internal `QueryOperation Listener` handler is registered with the `query` component that in turn calls the model methods).	Cancels any edits made in the dialog.

12.3 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 12.3.2, "How to Use a quickQuery Component Without a Model."

12.3.1 How to Add the quickQuery Component Using a Model

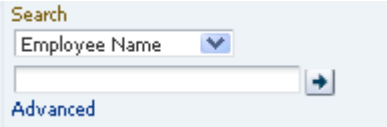
Before you begin

Implement a `QueryModel` class and associated classes. For more information, see Section 12.2, "Implementing the Model for Your Query."

To add a quickQuery component:

1. In the Component Palette, from the Common Components panel, drag a **Quick Query** and drop it onto the page.
2. Expand the **Common** section of the Property Inspector 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 12–2, or vertically as shown in Figure 12–5.

Figure 12–5 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 12.2, "Implementing the Model for Your Query."
   - **value**: Enter an EL expression that evaluates to the class that implements the `QueryDescriptor` class, as created in Section 12.2, "Implementing the Model for Your Query."
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 12.2, "Implementing the Model for Your Query."

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, add a command component to the `End` facet of the `quickQuery` component, and implement logic that will hide the `quickQuery` component and display the `query` component.

12.3.2 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.

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 Component Palette, from the Common Components panel, drag a **Quick Query** and drop it onto the page.
4. In the Property Inspector, 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 12–2, or vertically, as shown in Figure 12–5.
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 Component Palette, from the Common Components panel, drag a **Select One 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 Component Palette, add select list items as needed. For more information about using the `selectOneChoice` and `selectItems` components, see Section 9.6, "Using Selection Components."

9. In the Component Palette, from the Common Components 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, add a command component to the `End` facet of the `quickQuery` component, and implement logic that will hide the `quickQuery` component and display the `query` component.
11. In the Component Palette, from the Common Components 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.

12.3.3 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.

12.4 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 12–3.

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 12–3 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 12–4.

Table 12–4 Rendered Component for Search Criteria Field of Type Number

Operator	Component	Component When Multiple Select Is Enabled
Equals	Default list type hint	af:selectManyChoice
Does not equal	Default list type hint	af:selectManyChoice
Less than	af:inputNumberSpinBox	af:inputNumberSpinBox
Less than or equal to	af:inputNumberSpinBox	af:inputNumberSpinBox
Greater than	af:inputNumberSpinBox	af:inputNumberSpinBox
Greater than or equal to	af:inputNumberSpinBox	af:inputNumberSpinBox
Between	af:inputNumberSpinBox	af:inputNumberSpinBox
Not between	af:inputNumberSpinBox	af:inputNumberSpinBox
Is blank	None	None
Is not blank	None	None

If the underlying attribute is the `Date` data type, the component that will be rendered is shown in Table 12–5.

Table 12–5 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 11, "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 12–6 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 12–6 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 31, "Allowing User Customization on JSF Pages."

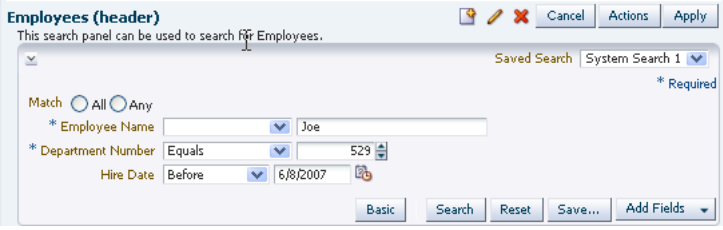
A seeded search is essentially a saved search that was created by the application developer. When the component is initialized, any seeded searches associated with that `query` component become available for the user to select.

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 31, "Allowing User Customization on JSF Pages."

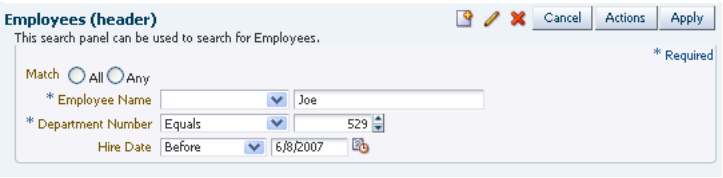
Along with the default display described previously, you can also configure the `query` component to display in a compact mode, simple mode, or design mode. The compact mode has no header or border, and the **Saved Search** dropdown list moves next to the expand or collapse icon. Figure 12–7 shows the same `query` component as in Figure 12–6, but set to compact mode.

Figure 12–7 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 12–8 shows the same `query` component set to simple mode.

Figure 12–8 Query Component in Simple Mode



The design mode has the same visual appearance as the simple mode but is used mostly for designing the `QueryDescriptor`.

The `query` component supports `toolbar` and `footer` facets that allow you to add additional components to the query, such as command 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.

12.4.1 How to Add the Query Component

Before you begin:

Implement a `QueryModel` class and associated classes. For more information, see Section 12.2, "Implementing the Model for Your Query."

To add a query component:

1. In the Component Palette, from the Common Components panel, drag a **Query** and drop it onto the page.
2. In the Property Inspector, 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 12.2, "Implementing the Model for Your Query."
   - **value**: Enter an EL expression that resolves to the `QueryDescriptor` class, as created in Section 12.2, "Implementing the Model for Your Query."

3. Expand the **Appearance** section and set the following attributes:
   - **displayMode**: Specify if you want the component to display in Default, Simple, Compact, or Design 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 12.2, "Implementing the Model for Your Query."
5. Expand the **Other** section and set the following:
   - **criterionFeatures**: Enter `matchCaseDisplayed` to allow the user to set `matchCase` for a criterion. This option is available only for `String` data types.
   - **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 Component Palette, from the Common Components 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.

13

Using Popup Dialogs, Menus, and Windows

This chapter describes how to create and use popup elements in secondary windows including dialogs, menus, and windows on JSF pages. The chapter also describes how to use the ADF Faces dialog framework to create dialogs with a separate page flow.

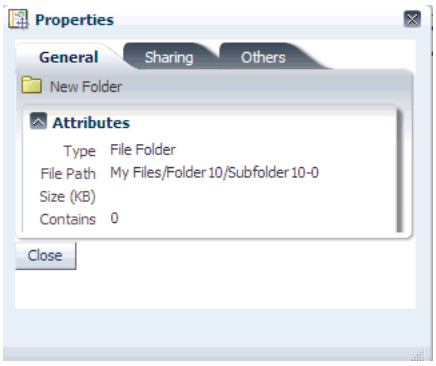
This chapter includes the following sections:

13.1 Introduction to Using Popup Elements

ADF Faces provides a set of rich client components for hiding and showing information in a secondary window. The `popup` component is an invisible layout control, typically used in conjunction with other components to display inline (that is, belonging to the same page) dialogs, windows, and menus.

For example, Figure 13–1 shows a dialog box created by placing a `dialog` component as a child to a `popup` component. A user can enter search criteria and click **OK** to submit the entry, or exit the dialog by clicking **Cancel** or closing the dialog.

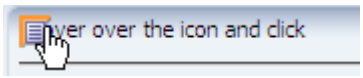
Figure 13–1 af:dialog Component



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 13–2.

Figure 13–2 With Mouseover, Larger Icon with Note is Displayed



When the user clicks the note icon, the associated popup displays its enclosed content.

ADF Faces also provides a *dialog framework* to support building pages for a process displayed *separate* from the parent page. This framework supports multiple dialog pages with a control flow of their own. For example, say a user is checking out of a web site 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 don't 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 Faces Controller for use with ADF task flows. For more information, see the "Running a Bounded Task Flow in a Modal Dialog" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

13.2 Declaratively Creating Popup Elements

The `dialog`, `panelWindow`, `menu`, and `noteWindow` components can all be used inside the `popup` component to display inline popup elements, as shown in Table 13–1. When no child component exists for the `popup` component, a very simple inline popup is displayed.

Table 13–1 Components Used with the popup Component

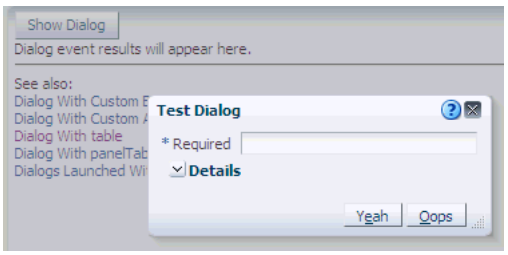
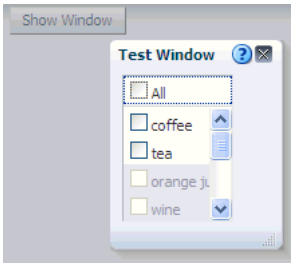
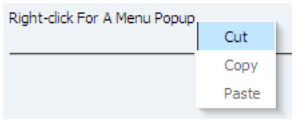
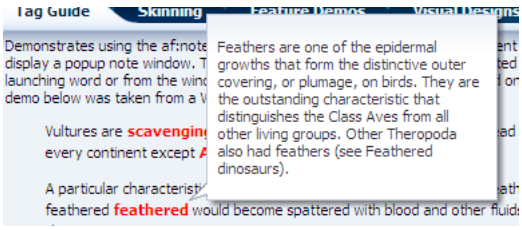
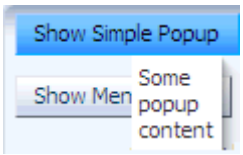
Component	Displays at Runtime
`dialog`	Displays its children inside a dialog and delivers events when the `OK`, `Yes`, No, and `Cancel` actions are activated. For more information, see Section 13.2.1, "How to Create a Dialog." 

Table 13–1 (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 13.2.2, "How to Create a Panel Window." 
`menu`	Displays a context menu for an associated component. For more information, see Section 13.2.3, "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 13.2.4, "How to Create a Note Window." 
`popup` component without a parent component	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). For more information, see Section 17.5, "Displaying Help for Components."

Typically, you use a command component in conjunction with the `showPopupBehavior` tag to launch a popup element. You associate the `showPopupBehavior` tag with the component it should launch. This tag also controls the positioning of the popup element (when needed).

In addition to being used with action events on command components, the `showPopupBehavior` tag can be used in conjunction with other events, such as the `showDetail` event and the `selection` event. For more information, see Section 13.4, "Invoking Popup Elements."

By default, the content of the popup element is not sent from the server until the popup element is displayed. This represents a trade-off between the speed of showing the popup element when it is opened and the speed of rendering the parent page. Once the popup element 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 element 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 popup elements that are consistently used by all users every time they use the page.
- `lazyUncached` - The content is not loaded until the popup element is displayed, and then the content is reloaded every time you show the popup element. Use this strategy if the popup element shows data that can become stale or outdated.

If you choose to set the `popup` component's `contentDelivery` attribute to `lazy` or `lazyUncached`, 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`.

13.2.1 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`. It is recommended that any of these buttons that you add have their `partialSubmit` attribute set to `true`. This makes sure that an `actionEvent` invokes only on components within the dialog. 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 command component'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 13.6, "Controlling the Automatic Cancellation of Inline Popups."

To create an inline dialog:

1. In the Component Palette, from the Common Components panel, drag and drop a **Popup** onto the JSF 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 Property Inspector, 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 13.2.5, "What Happens at Runtime: Popup Component Events."

3. Optionally, in the Property Inspector, expand the Other section and set a value for the **AutoCancel** property to determine the automatic cancel behavior. For more information, see Section 13.6, "Controlling the Automatic Cancellation of Inline Popups."
4. From the Component Palette, drag and drop a **Dialog** as a direct child to the `popup` component.
5. In the Property Inspector, expand the Common section and set the following attributes:

   - **Type**: Select the built-in partial-submit command buttons you want to display in your dialog.

     For example, if you set the `type` attribute to `yesNoCancel`, the dialog will display **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. 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 8.2.1, "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 will handle 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 13–1. 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 13–1 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 13–2 shows how the `dialogListener` attribute is bound to the `deleteItem` method.

Example 13–2 Binding the dialogListener attribute 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 5.3.5, "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, and any other events (including the `dialogEvent`) are prevented from getting to the server. However, 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 Other 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 31, "Allowing User Customization on JSF Pages."

   Note: If a command component without the `showPopupBehavior` tag is used to launch the dialog, and if that command component has values for the `windowHeight` and `windowWidth` attributes, the values on the command component will override the `contentHeight` and `contentWidth` values. For more information about the `showPopupBehavior` tag, see Section 13.4, "Invoking Popup Elements."

9. If needed, add command components to the `buttonBar` facet. It is recommended that you set the `partialSubmit` attribute to `true` for every added command component. However, you can set the command component's `partialSubmit`

attribute to `false` if the `af:popup` component's `autoCancel` property is set to `disabled`. The values an `af:popup` component's `autoCancel` property and a command component `partialSubmit` property determine how a command component dismisses and reloads a dialog. For more information, see Section 13.6, "Controlling the Automatic Cancellation of Inline Popups."

Tip: If the facet is not visible in the visual editor:

1. Right-click the `dialog` component in the Structure window.
2. From the context menu, choose **Facets - Dialog > ButtonBar**. Facets in use on the page are indicated by a checkmark in front of the facet name.

By default, added command components do not dismiss the dialog. You need to bind the `actionListener` on the command 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 13.4, "Invoking Popup Elements."

13.2.2 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 some logic to be invoked to handle data in the `panelWindow`, then 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 command buttons in the `bottom` facet.

To create an inline window:

1. In the Component Palette, from the Common Components panel, drag and drop a **Popup** onto the JSF 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 Property Inspector, 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"). You can override this setting by selecting `false`.
- **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**: Set to `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 13.2.5, "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.

3. Optionally, in the Property Inspector, expand the Other section and set a value for the **AutoCancel** property to determine the automatic cancel behavior. For more information, see Section 13.6, "Controlling the Automatic Cancellation of Inline Popups."
4. In the Component Palette, from the Layout panel, drag and drop a **Panel Window** as a direct child to the `popup` component.
5. In the Property Inspector, 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 8.2.1, "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.

6. If you want to set a fix size for the window, or if you have set **resize** to `on` or set **stretchChildren** to `first`, expand the Other 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 31, "Allowing User Customization on JSF Pages."

   Note: If a command component without the `showPopupBehavior` tag is used to launch the dialog, and if that command component has values for the `windowHeight` and `windowWidth` attributes, the values on the command component will override the `contentHeight` and `contentWidth` values. For more information about the `showPopupBehavior` tag, see Section 13.4, "Invoking Popup Elements."

7. Insert components to display or collect data for the window. Use a layout component like `panelGroupLayout` to contain the components.
8. Add logic on the parent page to invoke the popup and panel window. For more information, see Section 13.4, "Invoking Popup Elements."

13.2.3 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 14.3, "Using Toolbars."

To create an inline context menu:

1. In the Component Palette, from the Common Components panel, drag and drop a **Popup** onto the JSF 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 Property Inspector, 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 13.2.5, "What Happens at Runtime: Popup Component Events."

3. Optionally, in the Property Inspector, expand the Other section and set a value for the **AutoCancel** property to determine the automatic cancel behavior. For more information, see Section 13.6, "Controlling the Automatic Cancellation of Inline Popups."
4. From the Component Palette, drag and drop a **Menu** as a direct child to the `popup` component, and build your menu using `commandMenuItem` components, as documented starting with Step 6 in Section 14.2.1, "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 14.2.1, "How to Create and Use Menus in a Menu Bar."

5. Add logic on the parent page to invoke the popup and context menu. For more information, see Section 13.4, "Invoking Popup Elements."

13.2.4 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.

To create an inline window:

1. In the Component Palette, from the Common Components panel, drag and drop a **Popup** onto the JSF 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 Property Inspector, 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**: Set to `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 13.2.5, "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.

3. Optionally, in the Property Inspector, expand the Other section and set a value for the **AutoCancel** property to determine the automatic cancel behavior. For more information, see Section 13.6, "Controlling the Automatic Cancellation of Inline Popups."
4. From the Component Palette, drag and drop a **Note Window** as a direct child to the `popup` component.
5. 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`.

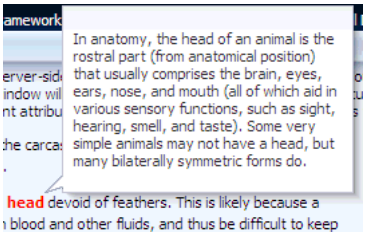
   Example 13–3 shows text for a note window.

Example 13–3 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 13–3 shows how the note would display.

Figure 13–3 Text Displayed in a Note Window



6. Optionally, in the Property Inspector, expand the Other section and specify a number of seconds for the **AutoDismissalTimeout** property. 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.

7. Add logic on the parent page to invoke the popup and note window. For more information, see Section 13.4, "Invoking Popup Elements."

13.2.5 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 command link. When the command link 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 command link that was clicked, you need to use the `eventContext` property to ensure that the context is that row, as shown in Example 13–4.

Example 13–4 Using eventContext for Shared Popup elements

```
<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:commandLink text="#{person.firstName}">
      <af:showPopupBehavior popupId="::noteWindow" triggerType="mouseHover"/>
    </af:commandLink>
  </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 13–5.

Example 13–5 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:commandLink text="#{person.firstName}">
      <af:showPopupBehavior popupId="::noteWindow" triggerType="mouseHover"/>
      <af:clientAttribute name="fullName" value="#{person.fullName}"/>
    </af:commandLink>
  </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 4.7.2, "How to Use the pageFlowScope Scope Without Writing Java Code." For more information about using client attributes, see Section 3.8, "Using Bonus Attributes for Client-Side Components." For more information about the `showPopupBehavior` tag, see Section 13.4, "Invoking Popup Elements."

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 3.2, "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 5.4, "Sending Custom Events from the Client to the Server."

13.3 Programmatically Invoking a Popup

You can programmatically show, hide, or cancel a popup in response to an `actionEvent` generated by a command 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 13.2, "Declaratively Creating Popup Elements" 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 13.2, "Declaratively Creating Popup Elements." 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 13–6.

Example 13–6 Binding a popup Component to a Backing Bean

```
<af:popup id="p1"
        binding="#{mybean.popup}"
        ...
        />
```

Once you have done this, you configure a command 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 13–7 shows a `showPopup` backing bean method that uses the `HINT_LAUNCH_ID` hint to identify the command component that passes the `actionEvent` to it and `p1` to reference the popup on which we invoke the `show` method.

Example 13–7 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 13–8 shows a backing bean method that cancels a popup in response to an `actionEvent`:

Example 13–8 Backing Bean Method Canceling a Popup

```
public void cancelPopupActionListener(ActionEvent event) {
   FacesContext context = FacesContext.getCurrentInstance();
   p1.cancel();
  }
```

Example 13–9 shows a backing bean method that hides a `popup` in response to an `actionEvent`:

Example 13–9 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 *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*.

13.3.1 How to Programatically Invoke a Popup

You configure the command component's `actionListener` attribute to reference the backing bean method that shows, cancels or hides the popup.

Before you begin:

Create the type of popup that you want the server-side method to invoke, as described in Section 13.2, "Declaratively Creating Popup Elements."

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 13.3, "Programmatically Invoking a Popup."

To programmatically invoke a popup:

1. In the Component Palette, from the General Controls panel, drag and drop a command component onto the JSF page.

   For example, a **Button** component.

2. In the Property Inspector, expand the Behavior section and set the following attributes:

   - **PartialSubmit**: set to `true` if you do not want the Fusion web application to render the entire page after an end user clicks the command component. The default value (`false`) causes the application to render the whole page after an end user invokes the command component. For more information about page rendering, see Chapter 7, "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 command component at runtime.

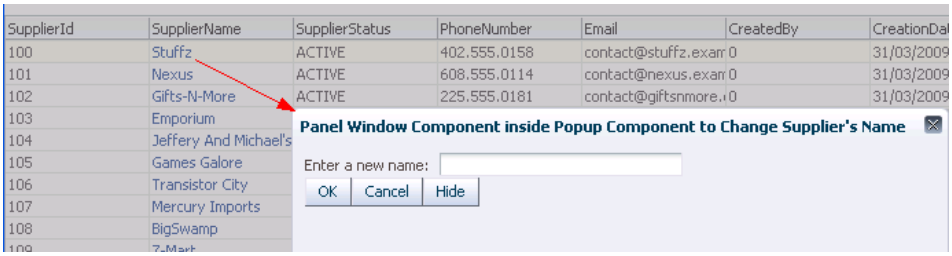
3. Write the logic for the backing bean that is invoked when the command component in step 1 passes an `actionEvent`.

   For more information, see Example 13–7, "Backing Bean Method Invoking a Popup", Example 13–8, "Backing Bean Method Canceling a Popup", or Example 13–9, "Backing Bean Method Hiding a Popup".

13.3.2 What Happens When You Programmatically Invoke a Popup

At runtime, end users can invoke the command components you configure to invoke the server-side methods to show, cancel, or hide a popup. For example, Figure 13–4 shows a `panelWindow` component that renders inside a `popup` component. It exposes two command buttons (**Cancel** and **Hide**) that invoke the `cancel` and `hide` methods respectively. End users invoke a `commandLink` component rendered in the **SupplierName** column of the `table` component in the underlying page to show the popup.

Figure 13–4 Popup Component Invoked by a Server-Side Method



13.4 Invoking Popup Elements

With ADF Faces rich client components, JavaScript is not needed to show or hide popups. The `showPopupBehavior` client behavior tag provides a declarative solution, so that you do not have to write JavaScript to open the popup or register the script with the component. For more information about client behavior tags, see Section 5.6, "Using Client Behavior Tags."

The `showPopupBehavior` tag listens for a specified event, for example the `actionEvent` on a command 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 to use a custom event as documented in Section 5.4, "Sending Custom Events from the Client to the Server."

13.4.1 How to Use the af:showPopupBehavior Tag

You use the `showPopupBehavior` tag in conjunction with the component that will invoke the popup element, for example a `commandButton` component that will invoke a dialog, or an `inputText` component that, when right-clicked, will invoke a context menu.

Before you begin:

1. Create a popup component and its holder.
2. Create the component that will invoke the popup.

To use the showPopupBehavior tag:

1. In the Component Palette, from the Operations panel, drag a **Show Popup Behavior** and drop it as a child to the component that will be used to invoke the popup element.
2. In the Property Inspector, use the dropdown menu for the **PopupId** attribute to choose **Edit**. Use the Edit Property: PopuId dialog to select the popup component to invoke.
3. From the **TriggerType** dropdown menu, choose the trigger that should invoke the popup. The default is `action` which can be used for command 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 `showPopupBehavior` tag documentation.

   Note: The event selected for the `triggerType` attribute will not be delivered to the server. If you need to invoke server-side logic based on this event, then you must launch the popup using either JavaScript or a custom event as documented in Section 5.4, "Sending Custom Events from the Client to the Server."

4. From the **AlignId** dropdown, choose **Edit**, and then use the Edit Property: AlignId dialog to select the component with which you want the popup to align.
5. From 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.

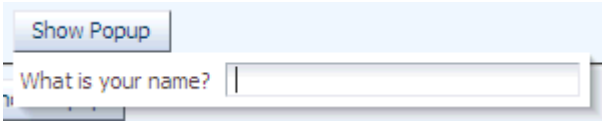
Example 13–10 shows sample code that displays some text in the `af:popup` component with the id "`popup1`" when the button "Click Me" is clicked.

Example 13–10 showPopupBehavior Associated with commandButton component

```
<af:commandButton text="Click me" id="button">
  <af:showPopupBehavior popupId="popup1" alignId="button" align="afterEnd"/>
</af:commandButton>

<af:popup id="popup1">
  <af:panelGroupLayout layout="vertical">
    <af:outputText value="Some"/>
    <af:outputText value="popup"/>
    <af:outputText value="content"/>
  </af:panelGroupLayout>
</af:popup>
```

The code in Example 13–10 tells ADF Faces to align the popup contents with the `commandButton` that is identified by the `id` *button*, and to use the alignment position of `afterEnd`, which aligns the popup element underneath the button, as shown in Figure 13–5.

Figure 13–5 Button and Popup Contents

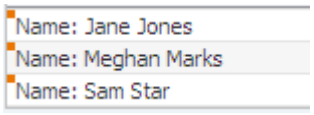


13.5 Displaying Contextual Information

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 element that was launched with a right-click on a component, the user would have no way of knowing the information was available in a popup.

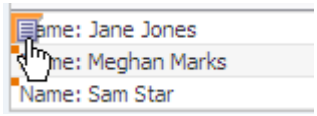
The `contextInfo` component allows you to display additional information in a popup element and also notifies users that additional information is available. When you place the `contextInfo` component into the context facet of a component that supports contextual information, a small orange square is shown in the upper left-hand corner of the component, as shown in Figure 13–6.

Figure 13–6 contextInfo Displays a Square



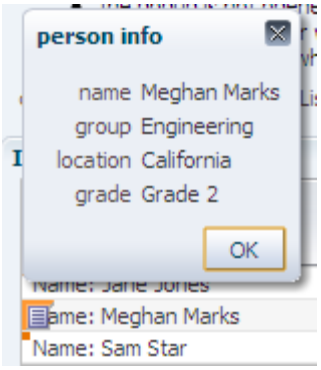
When the user places the cursor over the square, a larger triangle with a note icon and tooltip is displayed, indicating that additional information is available, as shown in Figure 13–7

Figure 13–7 contextInfo Component Indicates Additional Information Is Available



Because a `showPopupBehavior` tag is a child to the `contextInfo` component, the referenced popup will display when the user clicks the information icon, as shown in Figure 13–8.

Figure 13–8 Dialog launched From contextInfo Component



13.5.1 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`
   - `commandLink`
   - `inputComboboxListOfValues`
   - `inputListOfValues`
   - `inputText`
   - `outputFormatted`
   - `outputText`
   - `selectOneChoice`
2. Create the popup element to display, as documented in Section 13.2, "Declaratively Creating Popup Elements."

To use a contextInfo component:

1. In the Component Palette, from the Common Components 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:

   1. Right-click the `outputText` component in the Structure window.
   2. From the context menu, 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, bind the `contextInfoListener` attribute 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 5.4, "Sending Custom Events from the Client to the Server."

3. In the Component Palette, from the Operations panel, 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 Property Inspector, set the attributes as described in Section 13.4.1, "How to Use the af:showPopupBehavior Tag." For the **triggerType** value, be sure to enter `contextInfo`.

13.6 Controlling the Automatic Cancellation of Inline Popups

You can use the `af: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 13.2.1, "How to Create a Dialog."
- `panelWindow` component to create an inline window

  For more information, see Section 13.2.2, "How to Create a Panel Window."
- Menu components to create context menus

  For more information, see Section 13.2.3, "How to Create a Context Menu."
- `noteWindow` component to create a note window

  For more information, see Section 13.2.4, "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 a command component that has its `partialSubmit` property set to `false`. The Fusion web application renders the entire page after it invokes such a command component. In contrast, a command 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 7, "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 8.8, "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.

13.6.1 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 13.6, "Controlling the Automatic Cancellation of Inline Popups."

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 Property Inspector, expand the Other section and use the dropdown menu for the **AutoCancel** property to choose **disabled**.

13.6.2 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 13–11:

Example 13–11 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.

14

Using Menus, Toolbars, and Toolboxes

This chapter describes how to create menu bars and toolbars that contain tool buttons.

For information about creating navigation menus, that is, menus that allow you to navigate through a hierarchy of pages, see Section 18.5, "Using Navigation Items for a Page Hierarchy."

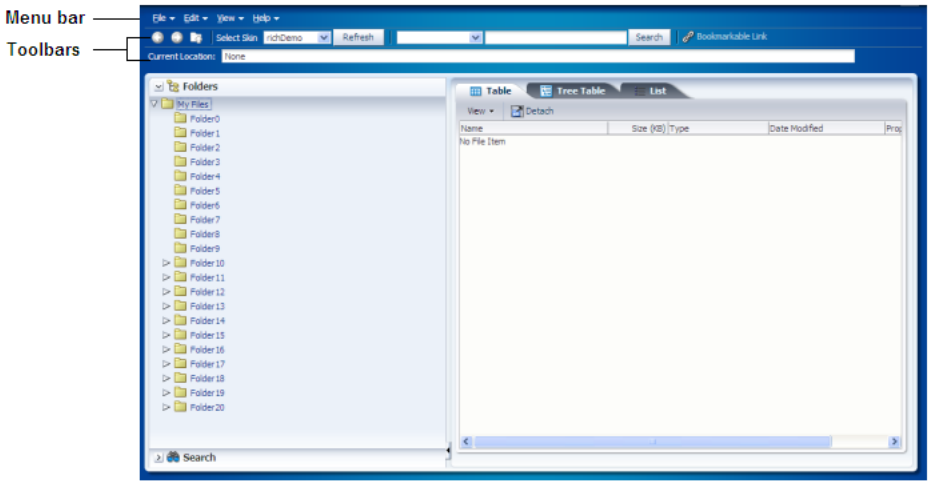
This chapter includes the following sections:

- Section 14.1, "Introduction to Menus, Toolbars, and Toolboxes"
- Section 14.2, "Using Menus in a Menu Bar"
- Section 14.3, "Using Toolbars"

14.1 Introduction to Menus, Toolbars, and Toolboxes

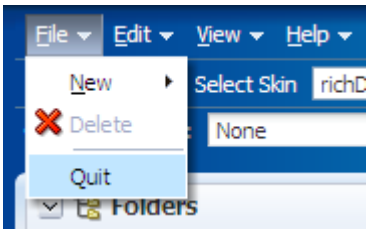
Menus and toolbars allow users to choose from a specified list of options (in the case of a menu) or to click buttons (in the case of a toolbar) to effect some change to the application. The File Explorer application contains both a menu bar and a toolbar, as shown in Figure 14–1.

Figure 14–1 Menu Bar and Toolbar in File Explorer Application



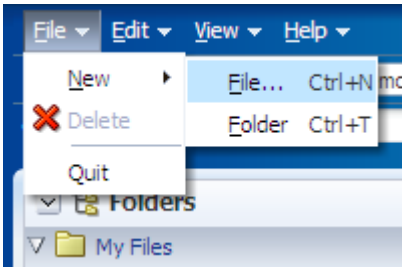
When a user chooses a menu item in the menu bar, the `menu` component displays a list of menu items, as shown in Figure 14–2.

Figure 14–2 Menu in the File Explorer Application



Note that as shown in Figure 14–3, menus can be nested.

Figure 14–3 Nested Menu Items



Buttons in a toolbar also allow a user to invoke some sort of action on an application or to open a popup 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.

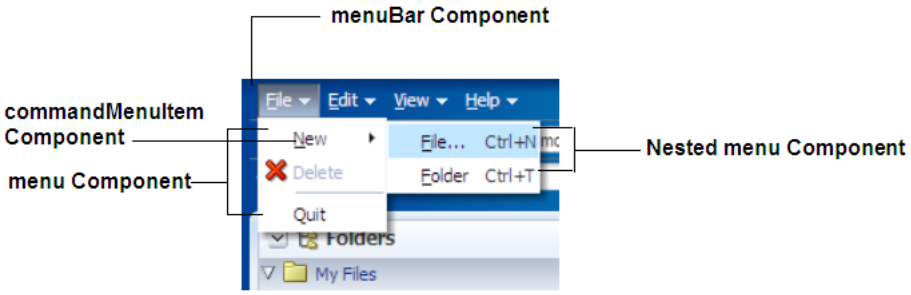
Note: If you want to create menus and toolbars in a table, then follow the procedures in Section 10.8, "Displaying Table Menus, Toolbars, and Status Bars."

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 13.2.3, "How to Create a Context Menu."

14.2 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). Each item on a menu bar is rendered by a `menu` component, which holds a vertical menu. Each vertical menu consists of a list of `commandMenuItem` components that can invoke some operation on the application. You can nest menu components inside menu components to create submenus. The different components used to create a menu are shown in Figure 14–4.

Figure 14–4 Components Used to Create a Menu



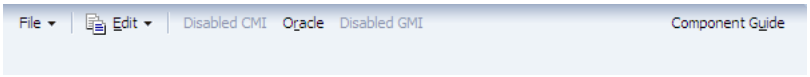
You can use more than one menu bar by enclosing them in a toolbox. Enclosing them in a toolbox stacks the menu bars so that the first menu bar in the toolbox is displayed at the top, and the last menu bar is displayed at the bottom. When you use more than one menu bar in a single toolbox row (by having them grouped inside the toolbox), then the `flex` attribute will determine which menu bar will take up the most space.

If you wish 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.

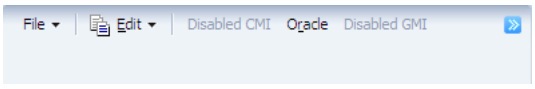
Within a menu bar, you can set one component to stretch so that the menu bar will always be the same size as its parent container. For example, in Figure 14–5, the menu bar is set to stretch a spacer component that is placed between the Disabled GMI menu and the Component Guide button. When the window 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 14–5 Spacer Component Stretches and Shrinks



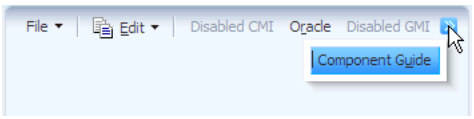
When a window is resized such that all the components within the menu bar can no longer be displayed, the menu bar displays an overflow icon, identified by the arrow cursor as shown in Figure 14–6.

Figure 14–6 Overflow Icon in a Menu Bar



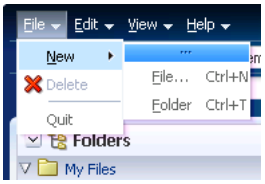
Clicking that overflow icon displays the remaining components in a popup window, as shown in Figure 14–7.

Figure 14–7 menu Component in an Overflow Popup Window



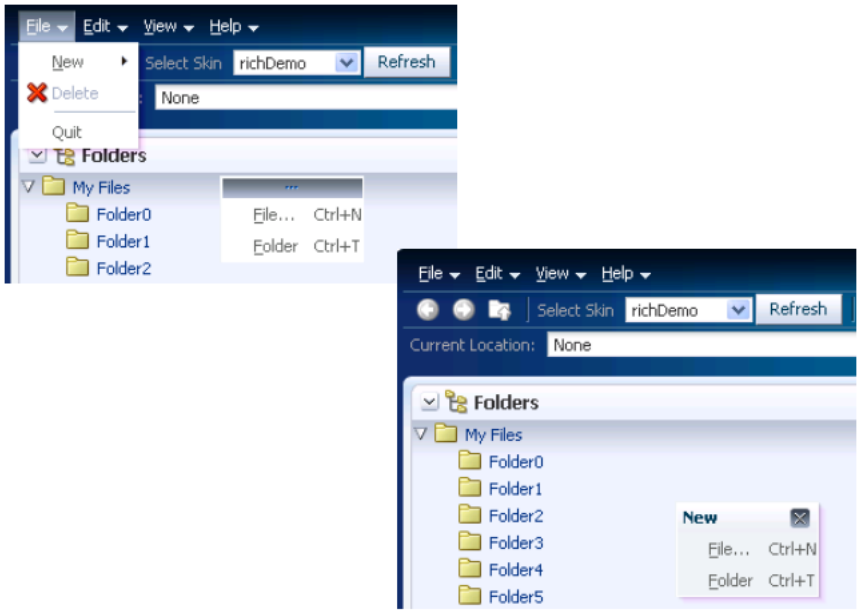
Menus and submenus can be made to be detachable and to float on the browser window. Figure 14–8 shows the **New** submenu in the **File** menu configured to be detachable. The top of the menu is rendered with a bar to denote that it can be detached.

Figure 14–8 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 14–9.

Figure 14–9 Floating Detached Menu



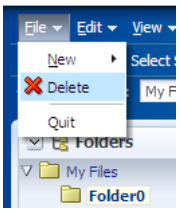
Tip: Consider using detachable menus when you expect users to:

- 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 14–10 shows the **Delete** menu item configured to display a delete icon.

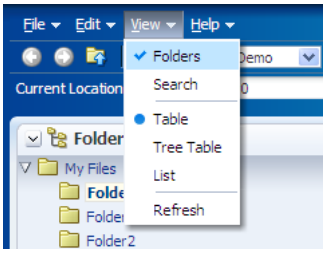
Note: Because alternative text cannot be provided for this icon, in order to create an accessible product, use this icon only when it is purely decorative. You must provide the meaning of this icon in some accessible manner.

Figure 14–10 Icons Can Be Used in Menus



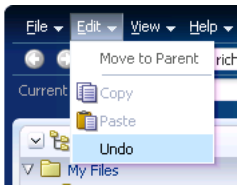
You can configure `commandMenuItem` components to be specific types that change how they are displayed when the menu item is chosen. For example, you can configure a `commandMenuItem` component to display a checkmark or a radio button next to the label when the item is chosen. Figure 14–11 shows the **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 14–11 Square Icon and Radio Button Denote the Chosen Menu Items



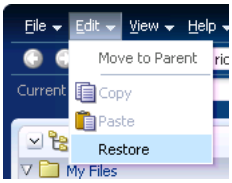
You can also configure a `commandMenuItem` component to have an antonym. Antonyms display different text when the user chooses a menu item. Figure 14–12 shows an **Undo** menu item in the **Edit** menu (added to the File Explorer application for this example).

Figure 14–12 The Edit Menu of the File Explorer Application



By configuring the `commandMenuItem` component for the **Undo** menu item to be an antonym and to have alternate text to display, when a user chooses **Undo**, the next time the user returns to the menu, the menu item will display the antonym **Restore**, as shown in Figure 14–13.

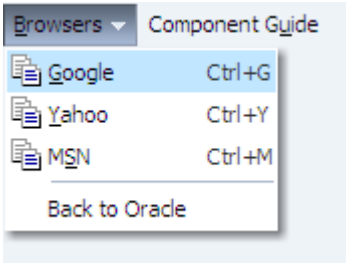
Figure 14–13 Menu Items Can Be Antonyms



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 `goLink` component, in that they perform direct page navigation, without delivering an `ActionEvent` event. Figure 14–14 shows three `goMenuItem` components used to navigate to external web sites.

Figure 14–14 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 13.2.3, "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 Section 5.6, "Using 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 18.6, "Using a Menu Model to Create a Page Hierarchy."

14.2.1 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 10.8, "Displaying Table Menus, Toolbars, and Status Bars."

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** component from the Layout panel of the Component Palette.

   > **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 **Panel Menu Bar** from the Common Components panel of the Component Palette. If you are using a toolbox component, the **Panel 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 14.3, "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 14–5, 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 14–1 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 14.3.2, "What Happens at Runtime: Determining the Size of Menu Bars and Toolbars."

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 14.3.2, "What Happens at Runtime: Determining the Size of Menu Bars and Toolbars."

4. Insert the desired number of `menu` components into the menu bar by dragging a **Menu** from the Component Palette, 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**. 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 Property Inspector and set the following attributes:

   - **Text**: Enter text for the menu's label. If you wish 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 22.3, "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.

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

6. If you want the menu to be detachable (as shown in Figure 14–8), expand the Behavior section in the Property Inspector 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 14–9).
7. If you want the menu to use lazy content delivery, expand the Other section in the Property Inspector and 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 invoke some sort of action along with navigation, drag a **MenuItem** from the Component Palette 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.

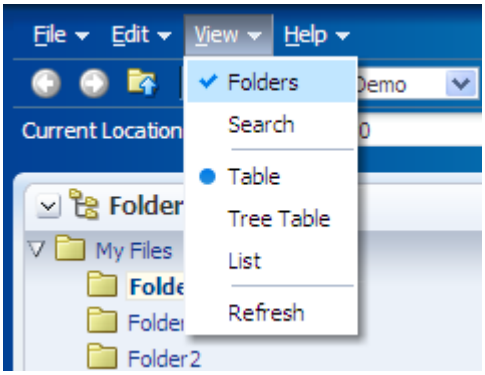
   If necessary, you can wrap the `commandMenuItem` components within a `group` component to display the items as a group. Example 14–2 shows simplified code for grouping the **Folders** and **Search** menu items in one group, the **Table**, **Tree Table** and **List** menu items in a second group, and the **Refresh** menu item by itself at the end.

Example 14–2 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 14–15 shows how the menu is displayed.

Figure 14–15 Grouped commandMenuItem Components in a Menu



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 20, "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 14–3).

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 Property Inspector 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:
     - **check**: In the default FusionFX skin, toggles a square 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.
     - **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**.
     - **default**: Assigns no type to this menu item. The menu item is displayed in the same manner whether or not it is chosen.
   - **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`.

Example 14–3 shows the **Special** menu with one group of menu items configured to use radio buttons and another group of menu items configured to show blue squares when chosen. The last group contains a menu item configured to be the antonym **Open** when it is first displayed, and then it toggles to **Closed**.

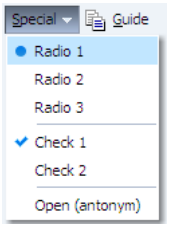
Example 14–3 Using the Type Attribute in a commandMenuItem Component

```
<af:menu text="Special">
  <af:group>
    <af:commandMenuItem text="Radio 1" type="radio" selected="true"
    <af:commandMenuItem text="Radio 2" type="radio"/>
    <af:commandMenuItem text="Radio 3" type="radio">
  </af:group>
  <af:group>
    <af:commandMenuItem text="Check 1" type="check" selected="true"
    <af:commandMenuItem text="Check 2" type="check"/>
  </af:group>
  <af:commandMenuItem text="Open (antonym)" type="antonym"
                      selectedText="Close (antonym)"/>
</af:menu>
```

Figure 14–16 shows how the menu will be displayed when it is first accessed.

Note: By default, ADF Faces components use the FusionFX skin, which displays the check type as a square. You can change this by creating your own skin. For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

Figure 14–16 Menu Items Using the Type Attribute



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.

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

- **Accelerator**: Enter the keystroke that will activate this menu item's command when the item is chosen, for example, `Control O`. ADF Faces converts the keystroke and displays a text version of the keystroke (for example, Ctrl+O) next to the menu item label, as shown in Figure 14–3.

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 22.3, "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 2.3, "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 **Go Menu Item** from the Component Palette as a child to the menu.

13. In the Property Inspector, expand the **Other** section and set the following attributes:

    - **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.

    - **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.

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

- **TargetFrame**: Use the dropdown list to specify where the new page should display. Values are
  - `_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.
- **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 22.3.4, "How to Define Access Keys for an ADF Faces Component."

14. If you want a menu bar to stretch so that it equals the width of the containing parent component, 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 14–7.

Example 14–4 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>
```

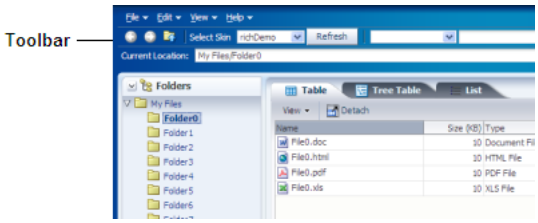
14.3 Using Toolbars

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 14–17 shows the toolbar from the File Explorer application.

> **Tip:** Toolbars can also include command buttons and command links (including the `commandImageLink` component) 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 14–17 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 also includes a `commandToolbarButton` 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.

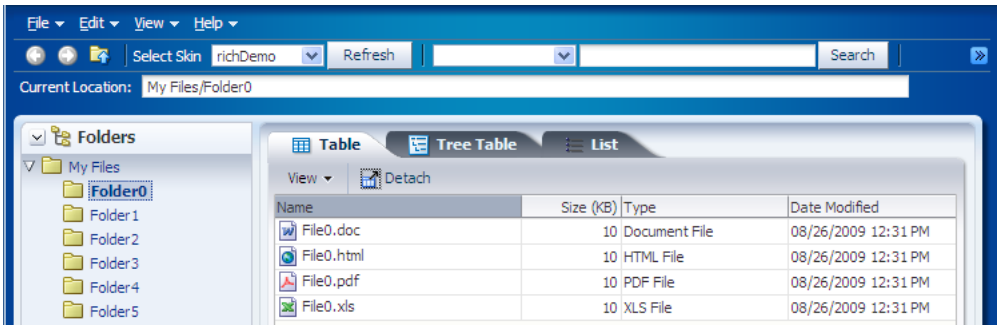
You can use more than one toolbar by enclosing them in a toolbox. Enclosing toolbars in a toolbox stacks them so that the first toolbar on the page is displayed at the top, and the last toolbar is displayed on the bottom. For example, in the File Explorer application, the currently selected folder name is displayed in the Current Location toolbar, as shown in Figure 14–17. 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.

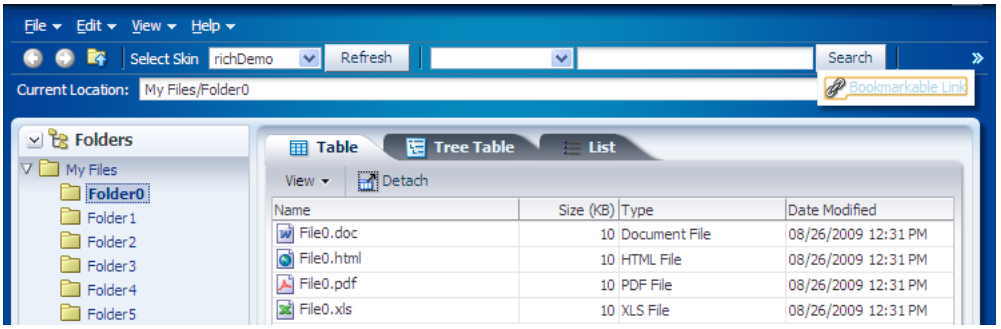
Within a toolbar, you can set one component to stretch so that the toolbar will always be the same size as its parent container. For example, in the File Explorer application, the lower toolbar that displays the current location contains the component that shows the selected folder. This component is 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 14–18.

Figure 14–18 Overflow Icon in a Toolbar



Clicking that overflow icon displays the remaining components in a popup window, as shown in Figure 14–19.

Figure 14–19 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.

14.3.1 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.

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 Layout panel of the Component Palette.

   > **Tip:** The `panelHeader`, `showDetailHeader`, and `showDetailItem` components support a `toolbar` facet for adding toolboxes and toolbars to section headers and accordion panel headers.

2. In the Component Palette, from the Common Components panel, 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 14–5, `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 14–5 Flex Attribute Determines Length of Toolbars

```
<af:toolbox>
  <af:toolbar id="toolbar1" flex="0">
    <af:commandToolbarButton text="ButtonA"/>
  </af:toolbar>
  <af:toolbar id="toolbar2" flex="2">
    <af:commandToolbarButton text="ButtonB"/>
  </af:toolbar>
  <af:toolbar id="toolbar3" flex="0">
    <af:commandToolbarButton text="ButtonC"/>
  </af:toolbar>
  <af:toolbar id="toolbar4" flex="1">
    <af:commandToolbarButton 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 14.3.2, "What Happens at Runtime: Determining the Size of Menu Bars and Toolbars."

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 14.3.2, "What Happens at Runtime: Determining the Size of Menu Bars and Toolbars."

4. Insert components into the toolbar as needed. To create a `commandToolbarButton` drag a **ToolbarButton** from the Component Palette 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 surrounding the group for the **Select Skin** dropdown list and **Refresh** button in Figure 14–17.

Toolbars 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 toolbar.

Tip: You can place other components, such as command 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.

Tip: If you plan to support changing the `visible` attribute of the button through active data (for example, data being pushed from the data source will determine whether nor not the toolbar is displayed), then you should use the `activeCommandToolbarButton` component instead of the `commandToolbarButton` component. Create an `activeCommandToolbarButton` component by dragging a **ToolbarButton (Active)** from the Component Palette.

5. For each `commandToolbarButton` component, expand the Common section of the Property Inspector and set the following attributes:
   - **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:
     - **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.

     - **default**: Assigns no type to this toolbar button.
   - **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**: 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.

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

- **Text**: Enter the label for this toolbar button.
- **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 2.3, "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 `commandToolbarButton` component. For information, see Section 14.2.1, "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 14–6.

Example 14–6 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 14–7.

Example 14–7 Using a Spacer to Justify Toolbar Components

```
<af:toolbar flex="1" stretchId="stretch1">
  <af:commandToolbarButton text="Forward"
                           icon="/images/fwdarrow_gray.gif"
                           disabled="true"></af:commandToolbarButton>
  <af:commandToolbarButton icon="/images/uplevel.gif" />

<!-- Insert a stretched spacer to push subsequent buttons to the right -->

  <af:spacer id="stretch1" clientComponent="true"/>

  <af:commandToolbarButton text="Reports" />
  <af:commandToolbarButton id="toggleRefresh"
                           text="Refresh:OFF" />
</af:toolbar>
```

14.3.2 What Happens at Runtime: Determining the Size of Menu Bars and Toolbars

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."

14.3.3 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.

15 Creating a Calendar Application

This chapter describes how to use the ADF Faces `calendar` component to create a calendar application.

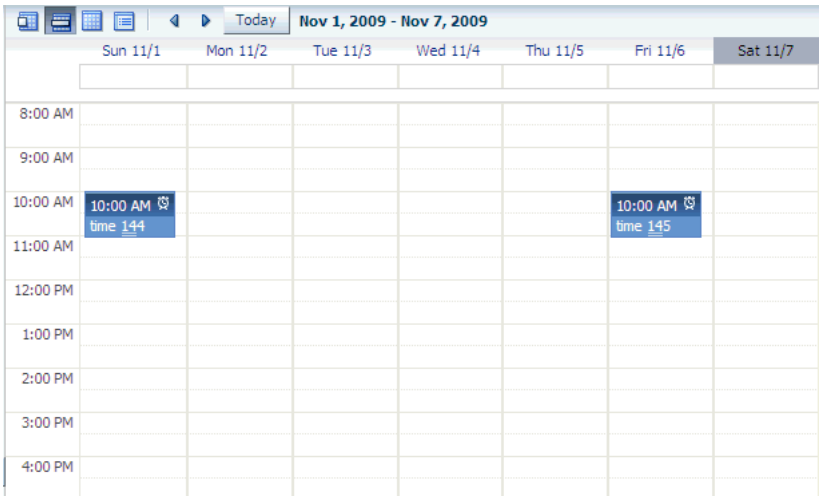
This chapter includes the following sections:

- Section 15.1, "Introduction to Creating a Calendar Application"
- Section 15.2, "Creating the Calendar"
- Section 15.3, "Configuring the Calendar Component"
- Section 15.4, "Adding Functionality Using Popup Components"
- Section 15.5, "Customizing the Toolbar"
- Section 15.6, "Styling the Calendar"

15.1 Introduction to Creating a Calendar Application

ADF Faces includes a calendar component that by default displays created activities in daily, weekly, monthly, or list views for a given provider or providers (a provider is the owner of an activity). Figure 15–1 shows an ADF Faces calendar in weekly view mode with some sample activities.

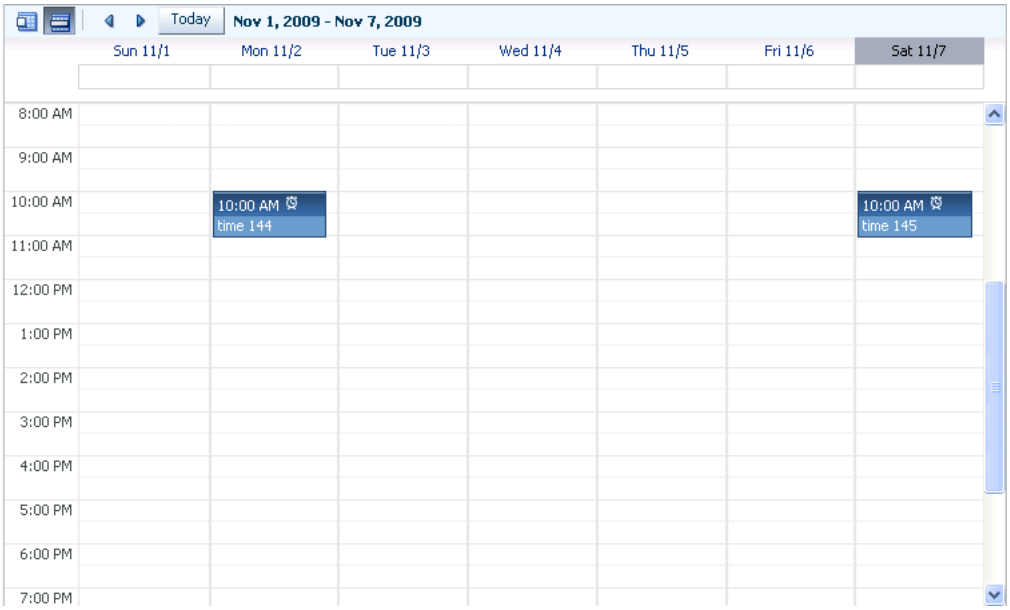
Figure 15–1 ADF Faces Calendar Showing Weekly View



You can configure the calendar so that it only displays a subset of those 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 15–2. Because only day and week views are available, those are the only buttons displayed in the toolbar.

Figure 15–2 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 (for example, Section A.6, "Configuration in trinidad-config.xml"). If a locale is not set 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 setting in `trinidad-config.xml`. You can override the default when you configure the calendar. For more information, see Section 15.3, "Configuring the Calendar Component."

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 of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 15.2, "Creating the Calendar."

The calendar includes a toolbar with built-in functionality that allows 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 15.5, "Customizing the Toolbar."

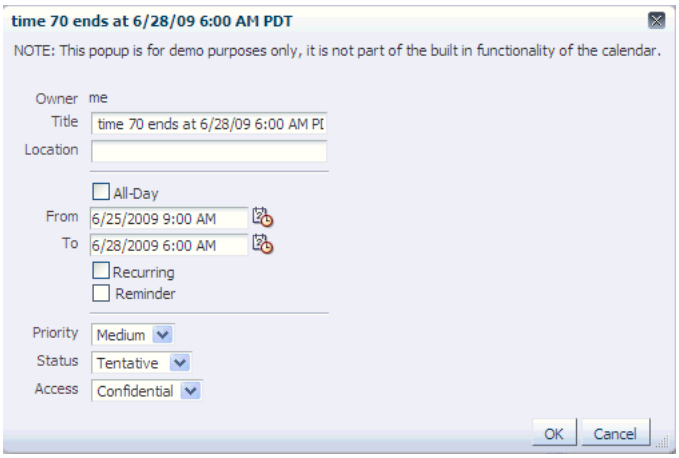
Tip: When these toolbar buttons are used, attribute values on the calendar are changed. You can configure these values to be persisted so that they remain for the user during the duration of the session. For more information, see Chapter 31, "Allowing User Customization on JSF Pages."

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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

The calendar component displays activities based on those 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 can allow 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 15–3.

Figure 15–3 Dialog Implemented to Edit an Activity



For more information about implementing additional functionality using events, facets, and popup components, see Section 15.4, "Adding Functionality Using Popup Components."

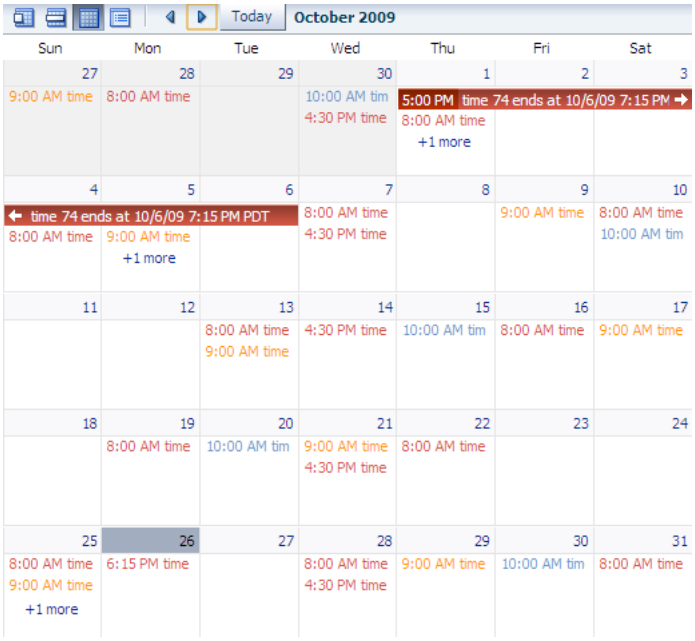
The calendar component supports the ADF Faces drag and drop architectural feature. Users can drag activities to different areas of the calendar, executing either a copy or a move operation, and can also drag handles on the activity to change the duration of the activity. For more information about adding drag and drop functionality, see Section 32.7, "Adding Drag and Drop Functionality to a Calendar."

By default, the calendar displays activities using a blue ramp. Color ramps are groups of colors all based on the same hue, for example, blue. 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 15–1. 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 15–4.

Figure 15–4 Month View with Activities from Different Providers



15.2 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. For an ADF Faces application, create the classes as managed beans. 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 allow you to declaratively create and bind your calendar component. For more information, see the "Using the ADF Faces Calendar Component" section of the *Oracle Fusion Middleware Fusion Developer's Guide for 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.

15.2.1 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, hour or minute). 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.

15.2.2 How to Create a Calendar

Create your own implementations of the `CalendarModel` and `CalendarActivity` classes and implement the abstract methods to provide the logic.

To create the calendar model classes:

1. Create a managed bean that will hold logic for the calendar. This bean must:
   - Extend the `oracle.adf.view.rich.model.CalendarModel` class.
   - Implement the abstract methods.

     For more information about the `CalendarModel` class, see the `ADF Faces Javadoc`.
   - 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 the demo application, see Section 1.4, "ADF Faces Demonstration Application").

   For more information about creating managed beans, see Section 2.6, "Creating and Using Managed Beans."
2. Create a managed bean that will hold logic for the activities. This bean must:
   - Extend the `oracle.adf.view.rich.model.CalendarActivity` class.
   - Implement the abstract methods.
   - Implement any other required functionality for the calendar. As 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 the `getTags` method must return a string that represents the activity instance. For more information, see Section 15.6.1, "How to Style Activities."

3. Create a managed bean that will hold information and logic for providers.
   - Extend the `oracle.adf.view.rich.model.CalendarProvider` class.
   - Implement the abstract methods.
   - Implement any other required functionality for the provider.

To create the calendar component:

1. In the Component Palette, from the Common Components section, drag a **Calendar** and drop it onto a 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 15.3, "Configuring the Calendar Component." For more information about stretching components, see Section 8.2.1, "Geometry Management and Component Stretching."

2. Expand the Calendar Data section of the Property Inspector, and enter an EL expression for **Value** that resolves to the managed bean that extends the `CalendarModel` class.

15.3 Configuring the Calendar Component

Configure the many display attributes for the calendar, for example, the day that a week starts, and the time displayed at the beginning of a day.

15.3.1 How to Configure the Calendar Component

You configure the calendar using the Property Inspector.

To configure a calendar:

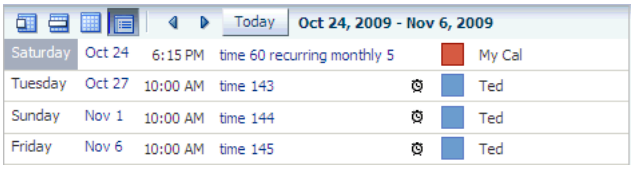
1. With the `calendar` component selected, expand the Common section of the Property Inspector, and set the following:
   - **View**: Select the view (either `day`, `list`, `month`, or `week`) that should be the default when the calendar is displayed. Users change this value when they click the corresponding button in the calendar's toolbar.
   - **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 (when in day or week view) starts the day at 12:01 a.m., the calendar will automatically scroll to the startHour value, so that it is displayed at the top of the view. The user can always 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 15–5 shows a calendar in list view with the listType set to dayCount and the listCount value set to 7.

Figure 15–5 List View Using dayCount Type



2. Expand the Calendar Data section of the Property Inspector, 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, February 6, 2009 is the active day. The month view will show February. If the user uses the next button to scroll to the next month, the active date will become March 6, 2009.
   - **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.
3. Expand the Other section of the Property Inspector and set **AvailableViews**. The value can be one of or a combination of the following:
   - month
   - week

- `day`
- `list`
- `all`

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 in the list.

If you do not enter `day` as an available view, then activities will be listed as plain text rather than as links in the `list` and `week` views (provided you do not also enter `all`).

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`.

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 32.7, "Adding Drag and Drop Functionality to a Calendar."

You can now add the following functionality:

- Allow users to create, edit, and delete activities using popup components. For more information, see Section 15.4, "Adding Functionality Using Popup Components."
- Allow users to move activities around on the calendar. For more information, see Section 32.7, "Adding Drag and Drop Functionality to a Calendar."
- Change or add to the toolbar buttons in the toolbar. For more information, see Section 15.5, "Customizing the Toolbar."
- Change the appearance of the calendar and events. For more information, see Section 15.6, "Styling the Calendar."

15.3.2 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 15.4, "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 dragging the box that displays the activity. The `CalendarDisplayChangeEvent` is fired whenever the component changes the value of a display attribute, for example when the `view` attribute changes from `month` to `day`.

When a `CalendarDisplayChangeEvent` is fired, the calendar component adds itself as a partial page rendering (PPR) target, allowing the calendar to be immediately refreshed. This is because the calendar assumes that if the display changed programatically, then the calendar must need to be rerendered. For example, if a user changes the `view` attribute from `day` to `month`, then the calendar is rerendered automatically.

15.4 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 allow users to create, delete, and edit activities, as well as to configure their instances of the calendar.

Table 15–1 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 15–1 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`

Table 15–1 (Cont.) Calendar Faces Events and Associated Facets

User Action	Event	Associated Facet	Component to Use in Popup
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`
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`

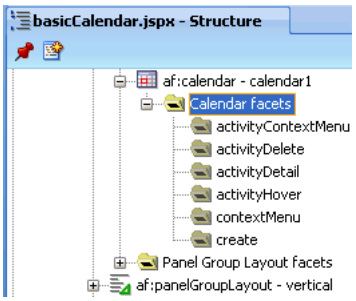
15.4.1 How to Add Functionality Using Popup Components

To add functionality, create the popups and associated components in the associated facets.

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 15–6.

Figure 15–6 Calendar Facets in the Structure Window



2. Based on Table 15–1, 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*** **>** ***ComponentName***.

   For more information about creating popup components, see Chapter 13, "Using Popup Dialogs, Menus, and Windows."

Example 15–1 shows the JSF code for a dialog popup component used in the `activityDelete` facet.

Example 15–1 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:outputText value="NOTE: This popup is for demo purposes only,
                  it is not part of the built in functionality of the calendar."/>
      <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 15–7 shows how the dialog is displayed when a user clicks an activity and presses the Delete key.

Figure 15–7 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 15–2 shows the `calendarActivityListener` for the calendar.jspx page in the ADF Faces demo application.

Example 15–2 calendarActivityLIstener Handler

```
  public void activityListener(CalendarActivityEvent ae)
  {

    CalendarActivity activity = ae.getCalendarActivity();

    if (activity == null)
    {
      // no activity with that id is found in the model
      System.out.println("No activity with event " + ae.toString());
      setCurrActivity(null);
      return;
    }

    System.out.println("providerId is " + activity.getProviderId());
    System.out.println("activityId is " + activity.getId());

    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 13, "Using Popup Dialogs, Menus, and Windows."

15.5 Customizing the Toolbar

By default, the toolbar in the calendar allows the user to change the view between day, week, month, and list, go to the next or previous item in the view, go to the present day, and also displays a text description of the current view, for example in the day view, it displays the active date, as shown in Figure 15–8.

Figure 15–8 Toolbar in Day View of a Calendar

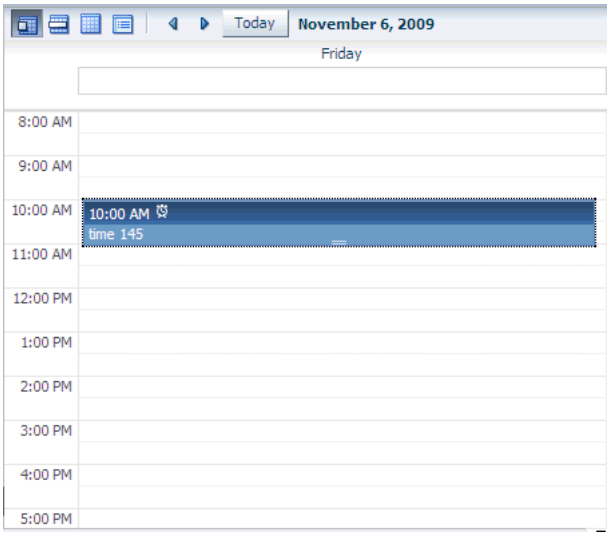
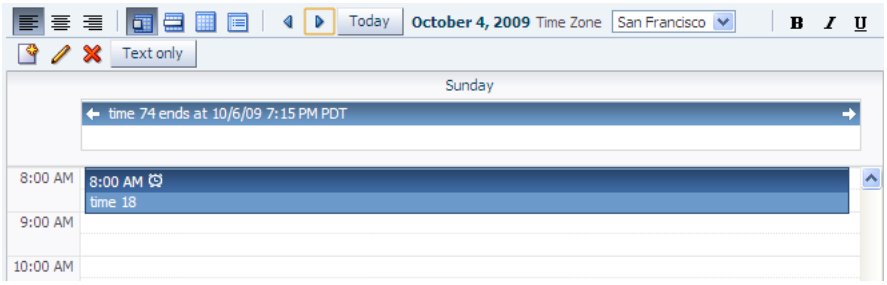


Figure 15–9 shows a toolbar that has been customized. It has added toolbar buttons, including buttons that are right-aligned on the top toolbar, and buttons in a second toolbar.

Figure 15–9 Customized Toolbar for a Calendar



15.5.1 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 toolbar, along with keywords that determine how or where the contained items should be displayed.

To customize the toolbar:

1. In the JSF page of the Component Palette, 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 15–9, 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 15–9 are in the `customToolbarAlign` facet.

2. In the ADF Faces page of the Component Palette, from the Common Components 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 14.3, "Using Toolbars."

3. In the Property Inspector, 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 15–3 shows the value of the `toolboxLayout` attribute for the toolbar displayed in Figure 15–9, 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 15–3 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:commandToolbarButton id="alignLeft" shortDesc="align left"
                               icon="/images/alignleft16.png" type="radio"
                               selected="true"/>
      <af:commandToolbarButton id="alignCenter" shortDesc="align center"
                               icon="/images/aligncenter16.png" type="radio"
                               selected="false"/>
      <af:commandToolbarButton id="alignRight" shortDesc="align right"
                               icon="/images/alignright16.png" type="radio"
                               selected="false"/>
    </af:toolbar>
```

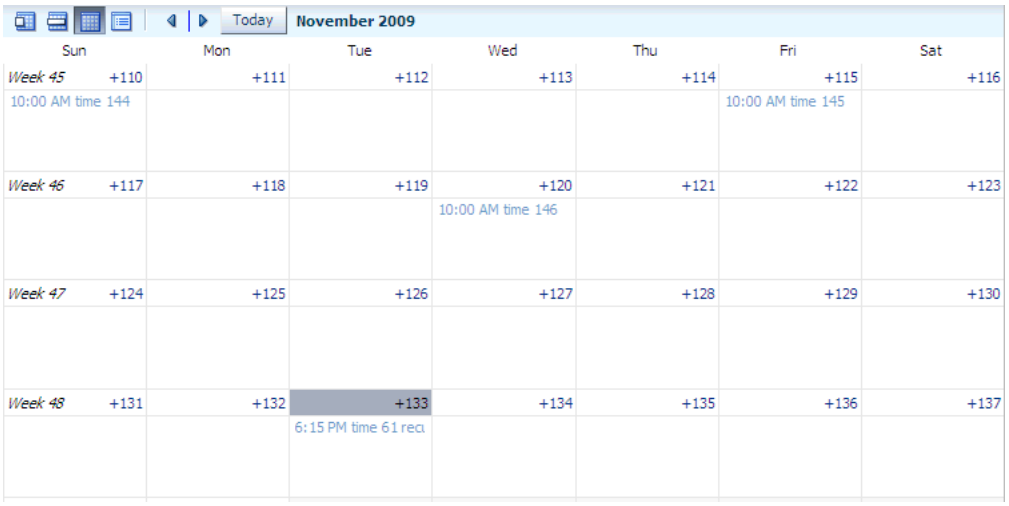
```
  </f:facet>
. . .
</af:calendar>
```

15.6 Styling the Calendar

Like other ADF Faces components, the calendar component can be styled as described in Chapter 20, "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 customize the display of dates. For example, you can display strings other than the calendar date for the day in the month view, allowing you to display countdown or countup type numbers, as shown in Figure 15–10.

Figure 15–10 Customized Display of Dates in a Calendar

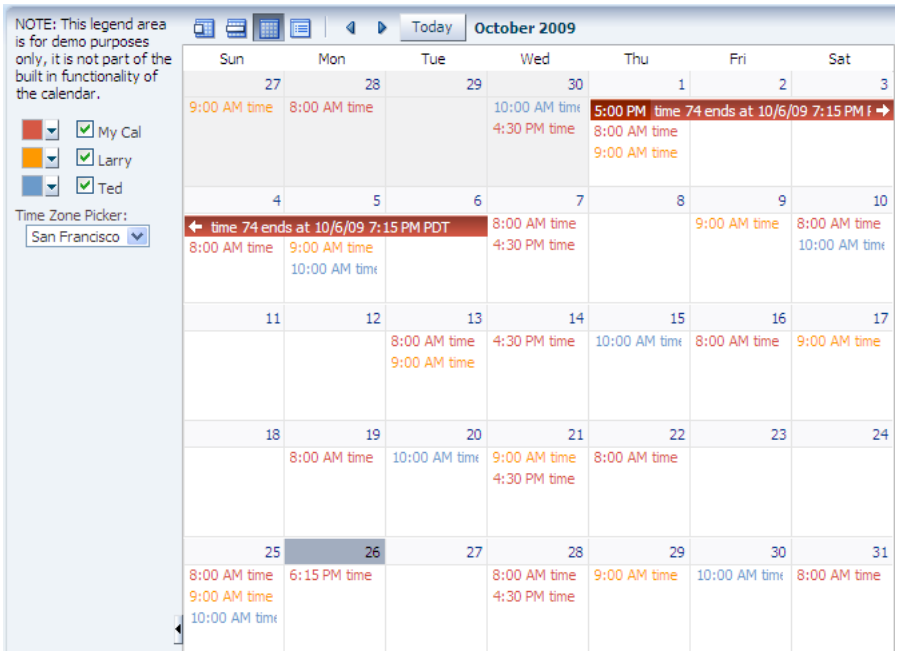


15.6.1 How to Style Activities

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 15–11 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 15–11 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 to represent the different states of activities. For example, T.F.'s activities use the Blue 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`, and can take a representative color (for example, the blue chosen for T.F.'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.

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 15–11, 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 L.E. might return `LE`, and an activity belonging to T.F. might return `TF`. For more information about implementing the `CalendarActivity` class, see Section 15.2.2, "How to Create a Calendar."

2. Create a map whose key is the string 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 15–11, you would create a map using the values shown in Table 15–2.

Table 15–2 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 Property Inspector, bind the `activityStyles` attribute to the map.

15.6.2 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.

15.6.3 How to Customize Dates

You can customize the display of dates in your calendar application. For example, 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.

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::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::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::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::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 15–4 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 15–10.

Example 15–4 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.

16 Using Output Components

This chapter describes how to display output text, images, and icons using ADF Faces components, and how to use components that allow users to play video and audio clips.

This chapter includes the following sections:

16.1 Introduction to 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.

Read-only text can be displayed using the `outputText` or `outputFormatted` components. The `outputFormatted` component enables you to add a limited set of HTML markup to the value of the component, allowing for some very simple formatting to the 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. You can display a collection of images in a carousel, which allows the users to spin through the collection to view each image.

The `media` component can play back an audio clip or a video clip. These components have attributes so that you can define how the item is to be presented on the page.

16.2 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, you 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 16–1 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 6.3, "Adding Conversion").

Example 16–1 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 16–2 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 16–2 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"/>
```

Figure 16–1 shows the different effects seen in a browser of the two different settings of the `escape` attribute.

Figure 16–1 Using the escape Attribute for Output Text

You should avoid setting the `escape` attribute to `false` unless absolutely necessary. A better choice is to use the `outputFormatted` component, which allows a limited number of HTML tags.

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 20, "Customizing the Appearance Using Styles and Skins."

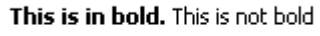
Example 16–3 shows an `outputFormatted` component displaying only a few words of its value in bold.

Example 16–3 Using outputFormatted to Bold Some Text

```
<af:outputFormatted value="&lt;b>This is in bold.&lt;/b> This is not bold"/>
```

Figure 16–2 shows how the component displays the text.

Figure 16–2 Text Formatted Using the outputFormatted Component



16.2.1 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.

To display output text:

1. In the Component Palette, from the Common Components panel, drag and drop an **Output Text** onto the page. To create an `outputFormatted` component, drag and drop an **Output Formatted** from the Component Palette.

   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 Component Palette.

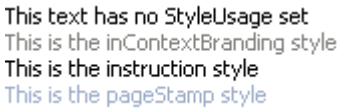
2. Expand the Common section of the Property Inspector and set the **value** attribute to the value to be displayed. If you are using the `outputFormatted` component, use HTML formatting codes to format the text as needed, as described in Table 16–1 and Table 16–2.

   The `outputFormatted` component also supports the `styleUsage` attribute whose values are the following predefined styles for the text:

   - `inContextBranding`
   - `instruction`
   - `pageStamp`

   Figure 16–3 shows how the `styleUsage` values apply styles to the component.

Figure 16–3 styleUsage Attribute Values



Note: If the `styleUsage` and `styleClass` attributes are both set, the `styleClass` attribute takes precedence.

16.2.2 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 16–1 lists the formatting codes allowed for formatting values in the `outputFormatted` component.

Table 16–1 Formatting Codes for Use in af:outputFormatted Values

Formatting Code	Effect
` `	Line break
`<hr>`	Horizontal rule
`<ol>...</ol><ul>...</ul><li>...</li>`	Lists: ordered list, unordered list, and list item
`<p>...</p>`	Paragraph
`<b>...</b>`	Bold
`<i>...</i>`	Italic
`<tt>...</tt>`	Teletype or monospaced
`<big>...</big>`	Larger font
`<small>...</small>`	Smaller font
`<pre>...</pre>`	Preformatted: layout defined by whitespace and line break characters preserved
`<span>...</span>`	Span the enclosed text
`<a>...</a>`	Anchor

Table 16–2 lists the character codes for displaying special characters in the values.

Table 16–2 Character Codes for Use in af:outputFormatted Values

Character Code	Character
`&lt;`	Less than
`&gt;`	Greater than
`&amp;`	Ampersand
`&reg;`	Registered
`&copy;`	Copyright
` `	Nonbreaking space
`&quot;`	Double quotation marks

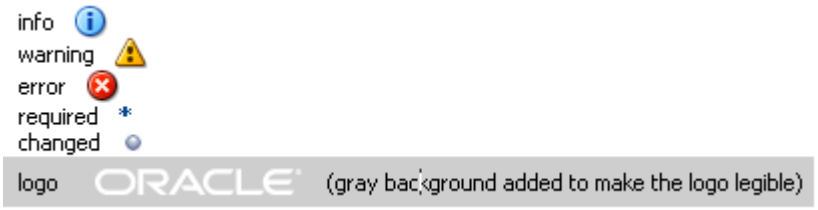
The attributes `class`, `style`, and `size` can also be used in the `value` attribute of the `outputFormatted` component, as can `href` constructions. All other HTML tags are ignored.

> **Note:** For security reasons, JavaScript is not supported in output values.

16.3 Displaying Icons

ADF Faces provides a set of icons used with message components, shown in Figure 16–4.

Figure 16–4 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 20, "Customizing the Appearance Using Styles and Skins."

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 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.

To display a standard icon:

1. In the Component Palette, from the Common Components panel, drag and drop an **Icon** onto your page.
2. Expand the Common section and set **Name** to the name of one of the icon functions shown in Figure 16–4. 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.

16.4 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, however, it must be placed inside a `goLink` component. For more information, see Section 16.5, "Using Images as Links."

To display an image:

1. In the Component Palette, from the Common Components panel, drag and drop an **Image** onto your 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 Component Palette.

2. In the Insert Image dialog, set the following:
   - **ShortDesc**: Set to the text to be used as the alternate text for the image.
   - **Source**: Enter the URI to the image file.
3. If you want to include a longer description for the image, in the Property Inspector, set **LongDescURL** attribute to the URI where the information is located.

16.5 Using Images as Links

ADF Faces provides the `commandImageLink` component that renders 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 `commandImageLink` component, see Section 18.2, "Using Buttons and Links for Navigation."

If you simply want an image to be used to navigate to a given URI, you can enclose the image in the `goLink` component and then, if needed, link to an image map.

You can use an image as a `goLink` component to one or more destinations. If you want to use an image as a simple link to a single destination, use a `goLink` component to enclose your image, and set the `destination` attribute of the `goLink` 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 `goLink` component and create a server-side image map for the image.

To use an image as one or more goLink components:

1. In the Component Palette, from the Common Components panel, drag and drop a **Go Link** onto the page.
2. Drag and drop an **Image** as a child to the `goLink` component.
3. In the Insert Image dialog, set the following:
   - **ShortDesc**: Set to the text to be used as the alternate text for the image.
   - **Source**: Enter the URI to the image file.
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 Property Inspector, set **ImageMapType** attribute to **server**.

- Select the **goLink** component and in the Property Inspector, 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 **goLink** component and enter the URI of the destination as the value of **Destination**.

16.6 Displaying Images in a Carousel

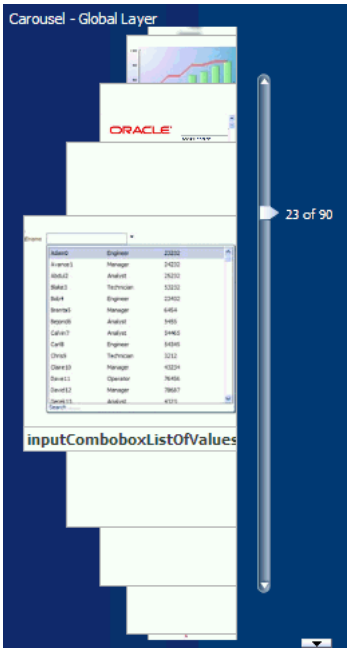
You can display images in a revolving carousel, as shown in Figure 16–5. Users can change the image at the front either by using the slider at the bottom or by clicking one of the auxiliary images to bring that specific image to the front.

Figure 16–5 The ADF Faces Carousel

By default, the carousel is displayed 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 card file. By default, the objects within the vertical orientation of the carousel are horizontally aligned to the center and the carousel itself is vertically aligned to the middle, as shown in Figure 16–6. You can change the alignments using the carousel's alignment attributes.

Figure 16–6 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.

The carousel component can display in circular mode, as in Figure 16–5, or you can configure it so that it displays only the current image, as shown in Figure 16–7

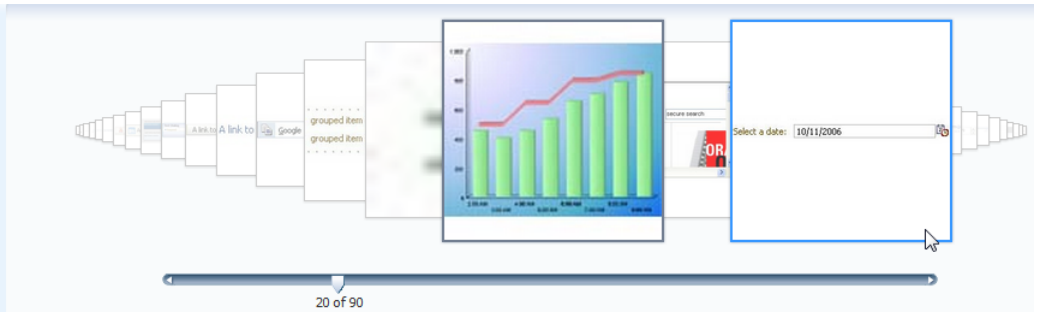
Figure 16–7 Carousel Can Display Just One Image.

You can also configure the controls used to browse through the images. You can display a slider that has next and previous arrows and that spans more than one image (as shown in Figure 16–5), you can display only next and previous buttons, (as shown in Figure 16–7), or you can display next and previous buttons, along with the slide counter, (as shown in Figure 16–8).

Figure 16–8 Next and Previous Buttons With a Slide Counter

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 Fusion FX v1.2 skin or later). You can configure the carousel so that instead, the item pops out and displays at full size, as shown in Figure 16–9.

Figure 16–9 Auxiliary Item Pops Out on Hover



When set to the circular mode, you can also configure the space between images, and you can also configure the size of the auxiliary images. By default, the space between images is set to 0.45 times the size of the preceding image, resulting in the images overlapping each other, and the auxiliary image size is set to 0.8, so that each image is 0.8th of the size as the preceding image, as shown in Figure 16–5. You can change these settings to alter how the carousel appears. For example, if you wanted the carousel to appear more like a filmstrip, you might set the space between the images to be 1.1, and the size of the auxiliary items to be 1, so that they are all the same size, as shown in Figure 16–10.

Figure 16–10 Configuring a Carousel to Display Like a Filmstrip

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 by an EL expression that uses 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 the parent component to the image displayed for each item.

For example, the `carouselItem` JSF page in the ADF Faces demo shown in Figure 16–5 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 the `image` component to retrieve the correct values for each item. Example 16–4 shows the JSF page code for the carousel. For more information about stamping behavior in a carousel, see Section 10.5, "Displaying Data in Trees."

Example 16–4 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

The carousel components are *virtualized*, meaning that not all the items available to 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 with 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 is 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.

16.6.1 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.

To Create a Carousel:

1. 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.

2. In the Component Palette, from the Common Components panel, drag and drop a **Carousel** onto the page.

3. In the Property Inspector, 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 16–6. 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:
     - **Bottom**: Aligns the items so that the bottom edges line up.
     - **Middle**: Aligns the items so that they have the same middle point. This is the default.
     - **Top**: Aligns the items so that the top edges line up.
   - **Value**: Bind the carousel to the model.

4. 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.
5. Expand the Appearance section and set EmptyText to the text that should display if no items are returned. If using a resource bundle, use the dropdown menu to choose **Select Text Resource**.
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 16–5 shows the handler method on the `CarouselBean` which redraws the detail panel when the spin happens.

Example 16–5 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**: Set to a number to determine how much an image will be offset from the preceding image. The default is `0.45`.
   - **AuxiliaryPopOut**: Set to `hover` to cause an auxiliary image to render full-size when the user hovers over it. The default is `off`.
   - **AuxiliaryScale**: Set to a number to determine what size each image should be in comparison to the image before it. A setting of `1` means all images would be the same size. A setting of less than `1` causes each image to be incrementally smaller, greater than `1` and they will be larger. By default, the setting is `0.8`, which means each image is 80% smaller than the preceding image.

- **ControlArea**: Specify the controls used to browse through the carousel images. Valid values are:
  - **full**: The slider is larger than the current image, and displays next and previous buttons.
  - **small**: The slider is the size of the current image, and displays next and previous buttons.
  - **compact**: Only the next and previous buttons are displayed.
  - **none**: The slider and controls are not displayed.
- **DisplayItems**: Select **circular** to have the carousel display multiple images. Select **oneByOne** to have the carousel display one image at a time.

9. From the Component Palette, drag a **Carousel Item** to the `nodeStamp` facet of the `Carousel` component.

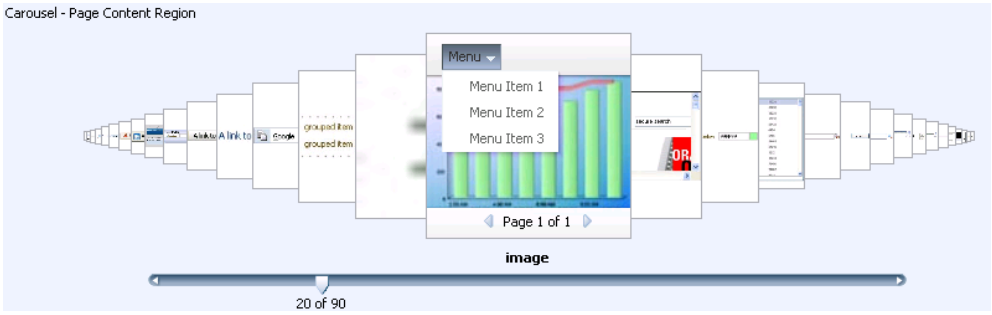
   Bind the `CarouselItem` component's attributes to the properties in the data model using the variable value set on the carousel's `var` attribute. For example, the carousel in Example 16–4 uses `item` as the value for the `var` attribute. So the value of the `carouselItem`'s `text` attribute would be `item.title` (given that `title` is the property used to access the text used for the carousel items on the data model).

10. Drag an image from the Component Palette 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 16–4 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 16–11 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 16–11 Using a More Complex Layout in a Carousel



Example 16–6 shows the corresponding page code.

Example 16–6 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">
      <f:facet name="separator">
        <af:spacer id="footSp" width="8"/>
      </f:facet>
      <af:commandImageLink . . .
                           />
      <af:outputText id="pageInfo" value="Page 1 of 1"/>
      <af:commandImageLink . . .
                          />
      </af:panelGroupLayout>
    </af:panelGroupLayout>
  </af:panelGroupLayout>
</af:carouselItem>
```

Performance Tip: The simpler the structure for the carousel is, the faster it will perform.

16.6.2 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 16–12.

Figure 16–12 Carousel Component Displayed in Google Chrome

Figure 16–13 shows the same component in Internet Explorer.

Figure 16–13 Carousel Component Displayed in Microsoft Internet Explorer

16.7 Displaying Application Status Using Icons

ADF Faces provides the `statusIndicator` component that 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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 16–3 shows the icons that are used to display server states for push and poll modes (note that the icons are actually animated).

Table 16–3 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 status indicator 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 20, "Customizing the Appearance Using Styles and Skins."

To use the status indicator icon:

1. In the Component Palette, from the Common Components panel, drag and drop a **Status Indicator** onto the page.
2. Use the Property Inspector to set any needed attributes.

Tip: For help in setting attributes, use the field's dropdown menu to view a description of the attribute.

16.8 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.

16.8.1 How to Allow Playing of Audio and Video Clips

Once you add a media component to your page, you can configure the media player to use by default, the size of the player and screen, the controls, and whether or not the clip should replay.

To include an audio or video clip in your application page:

1. In the Component Palette, from the Common Components 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 Property Inspector 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 Property Inspector 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.

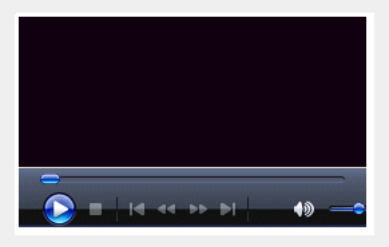
As an example, Example 16–7 uses the `all` setting for a `media` component.

Example 16–7 Controls for a Media Player

```
<af:media source="/images/myvideo.wmv" controls="all"/>
```

Figure 16–14 shows how the player is displayed to the user.

Figure 16–14 Media Player with All Controls



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 16–8 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 16–8 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"/>
```

17

Displaying Tips, Messages, and Help

This chapter describes how to define and display tips and messages for ADF Faces components, and how to provide different levels of help information for users.

This chapter includes the following sections:

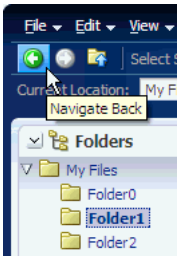
- Section 17.1, "Introduction to Displaying Tips and Messages"
- Section 17.2, "Displaying Tips for Components"
- Section 17.3, "Displaying Hints and Error Messages for Validation and Conversion"
- Section 17.4, "Grouping Components with a Single Label and Message"
- Section 17.5, "Displaying Help for Components"

17.1 Introduction to 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.

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 17–1 shows a tip configured for a toolbar button. For more information about creating tips, see Section 17.2, "Displaying Tips for Components."

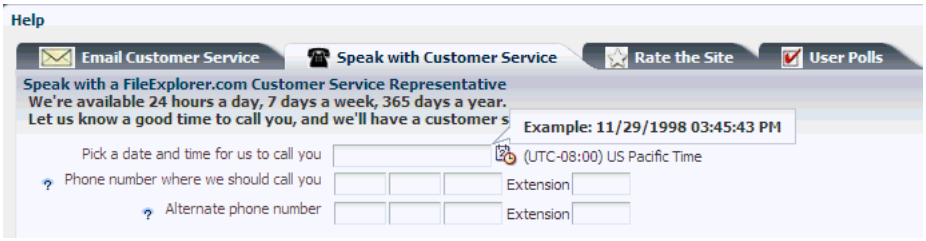
Figure 17–1 Tip Displays Information



Along with tips, `EditableValueHolder` components (such as the `inputText` component, or the selection components) can display hints used for validation and conversion. When you configure validation or conversion, a default hint automatically displays in a note window (for more information, see Chapter 6, "Validating and Converting Input"). 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 17–2. 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.

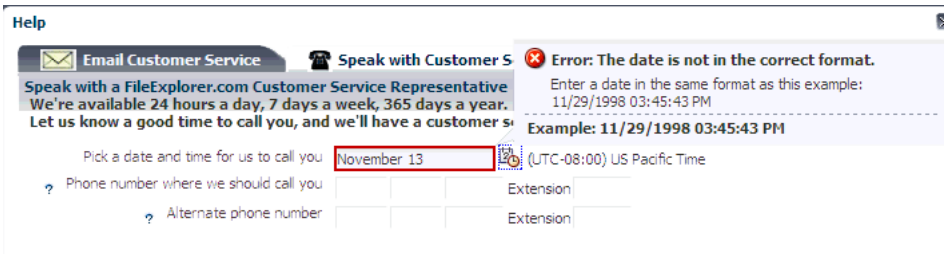
Figure 17–2 Attached Converters and Validators Include Messages



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. In general there are two types of messages that can be created: component-level messages, which 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 the 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.

For example, if a user enters a date incorrectly in the field shown in Figure 17–2, an error message is displayed, as shown in Figure 17–3. Note that the error message appears in the note window along with the hint.

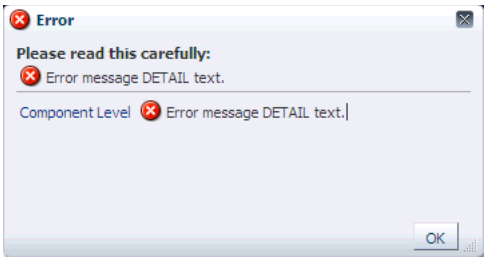
Figure 17–3 Validation and Conversion Errors Display in Note Window



If you want to display a 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.

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 two component messages), all messages in the queue will be displayed in a popup window, as shown in Figure 17–4.

Figure 17–4 Global and Component Messages Displayed by the Document



However, you can use the ADF Faces `messages` component if you want messages to display on the page rather than in a popup window. For more information about displaying hints and messages for components, see Section 17.3, "Displaying Hints and Error Messages for Validation and Conversion."

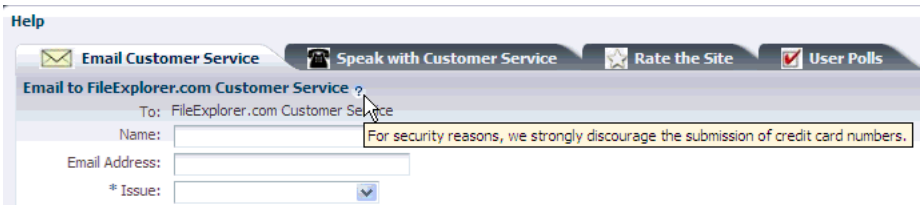
> **Tip:** While ADF Faces provides messages for validation and conversion, you can add your own `FacesMessages` objects to the queue using the standard JSF messaging API. When you do so, ADF Faces will display icons with the message based on the message level, as follows:

Instead of having each component display its own messages, you can use the `panelLabelAndMessage` component to group components and display a message in one area. This can be very useful when you have to group components together. For example, the File Explorer application uses a `panelLabelAndMessage` component where users enter a telephone number. The telephone number input field is actually three separate `inputText` components. The `panelLabelAndMessage` component wraps three `inputText` components. Instead of each having its own label and message, the three have just one label and one message, as shown in Figure 17–3. For more information, see Section 17.4, "Grouping Components with a Single Label and Message."

Instead of 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:

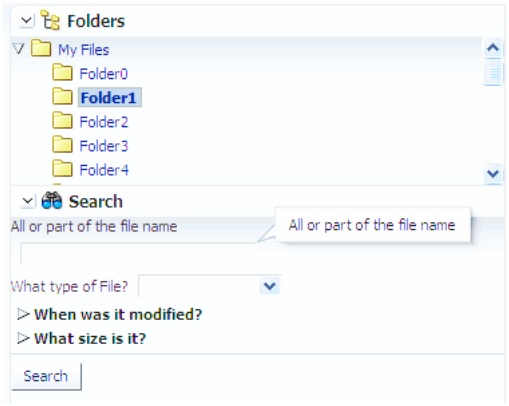
- Definition: Provides a help icon (question mark in a blue circle) with the help text appearing when the user mouses over the icon, as shown in Figure 17–5.

Figure 17–5 Definition Messages Display When Mousing Over the Icon



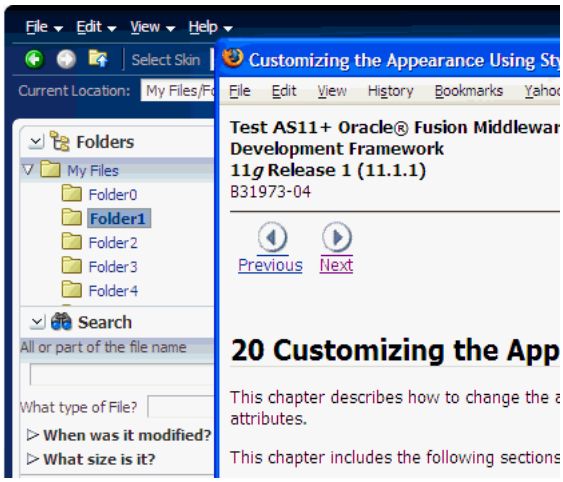
- Instruction: Depending on the component, this type of help either provides instruction text within the component (as with `panelHeader` components), or displays text in the note window that is opened when the user clicks in the component, as shown in Figure 17–6. The text can be any length.

Figure 17–6 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 17–7 shows the Select Skin `selectOneChoice` component configured to open a help topic about skins. When a user clicks the help icon, the help topic opens.

Figure 17–7 External URL Help Opens in a New Window



For more information about creating help systems, see Section 17.5, "Displaying Help for Components."

17.2 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 17.5, "Displaying Help for Components."

Figure 17–8 shows the effect when the cursor hovers over an `inputText` component.

Figure 17–8 Tip for an inputText Component

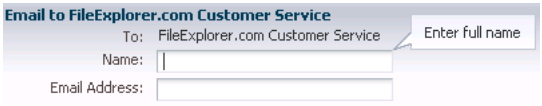
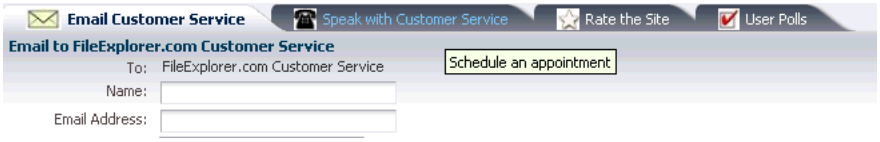


Figure 17–9 shows a tip as displayed for a `showDetailItem` component.

Figure 17–9 Tip for a showDetailItem Component



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 Property Inspector, 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.

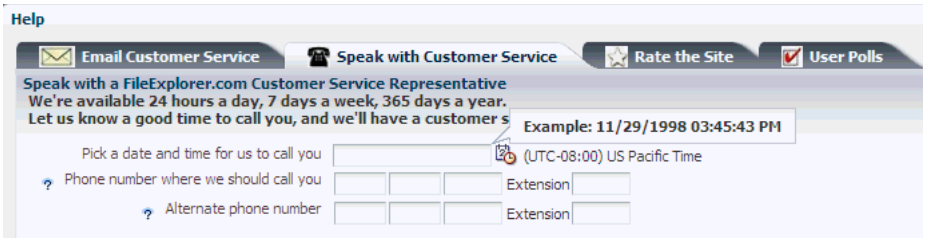
   If the text to be used is stored in a resource bundle, use the dropdown list to select **Select Text Resource**. Use the Select Text Resource dialog to either search for appropriate text in an existing bundle, or to create a new entry in an existing bundle. For more information about using resource bundles, see Chapter 21, "Internationalizing and Localizing Pages."

17.3 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. 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 17–10.

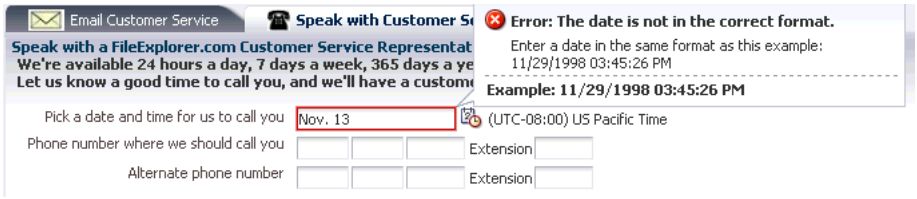
Figure 17–10 Validators and Converters Have Built-in Messages



When the value of an ADF Faces component fails validation, or cannot be converted by a converter, the component displays the resulting `FacesMessage` instance.

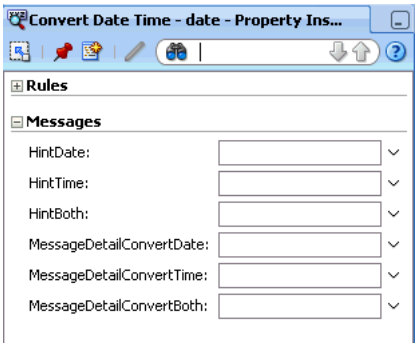
For example, entering a date that does not match the `dateStyle` attribute of the converter results in an error message, as shown in Figure 17–11.

Figure 17–11 Validation Error at Runtime



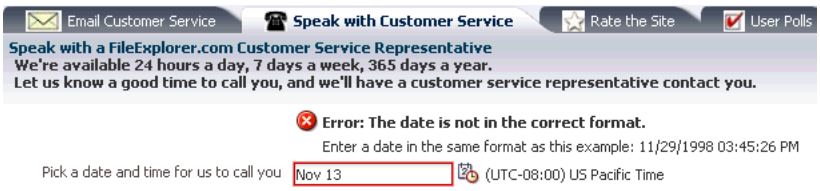
You can override the default validator and converter hint and error messages. Each ADF Faces validator and converter component has attributes you can use to define the detail messages to be displayed for the user. The actual attributes vary according to the validator or converter. Figure 17–12 shows the attributes that you can populate to override the messages for the `convertDateTime` converter, as displayed in the Property Inspector.

Figure 17–12 Message Attributes on a Converter



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 17–13 shows how messages are displayed using the `message` component.

Figure 17–13 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>`

17.3.1 How to Define Custom Validator and Converter Messages

To override the default validator and converter messages, set values for the different message attributes.

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.

Note: You can override messages only for ADF Faces components. If you want to create a message for a non-ADF Faces component (for example for the `f:validator` component), then use the `message` component. For more information, see Section 17.3.3, "How to Display Component Messages Inline."

2. In the Property Inspector, 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:

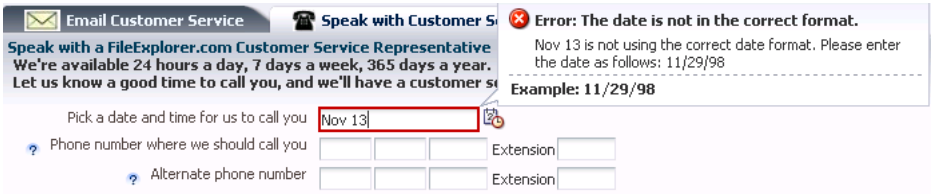
   - {0} the label that identifies the component
   - {1} the value entered by the user
   - {2}an example of the format expected by the component.

   Using these parameters, you could create this message:

   ```
   {1} 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 17–14.

Figure 17–14 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 21, "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 17.3.2, "What You May Need to Know About Overriding Default Messages Globally."

17.3.2 What You May Need to Know About Overriding Default Messages Globally

Instead of changing the message string per component instance with the `messageDetail[XYZ]` attributes, override the string globally so that the string will be displayed for all instances. To override globally, create a message bundle whose contents contain the key for the message and the message text you wish to use.

You create and use a message bundle in the same way you create and use resource bundles for translation, using either Java classes or properties files. For procedures and information, see Chapter 21, "Internationalizing and Localizing Pages."

For message key information, see Appendix B, "Message Keys for Converter and Validator Messages."

17.3.3 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.

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 Component Palette, from the Common Components 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.

17.3.4 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.

1. In the Component Palette, from the Common Components panel, drag a **Messages** and drop it onto the page where you want the messages to be displayed.
2. In the Property Inspector 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.

17.4 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 17–1 shows the JSF code for the `panelLabelAndMessage` component.

Example 17–1 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 17–15 shows how the `panelLabelAndMessage` and nested components are displayed in a browser.

Figure 17–15 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 17–16 shows how the telephone number fields would be displayed if the `End` facet was populated with an `outputText` component.

Figure 17–16 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 8.12, "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 8.6, "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.

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 and/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 Property Inspector, 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 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.

   > **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.

17.5 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.

The first type of external help provided by ADF Faces is Definition help. Like a standard tip, the content appears in a message box. However, instead of appearing when the user mouses over the component, Definition help provides a help icon (a blue circle with a question mark). When the user mouses over the icon, the content is displayed, as shown in Figure 17–17.

Figure 17–17 Definition Text for a Component

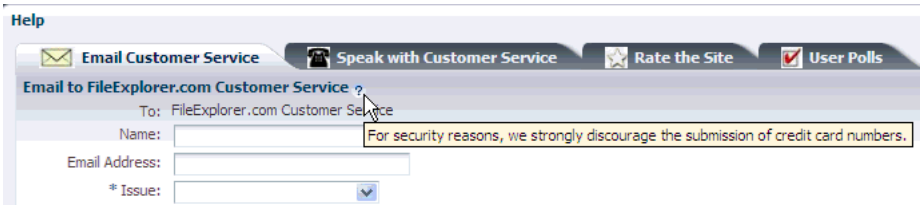
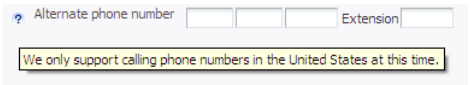
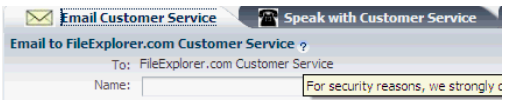
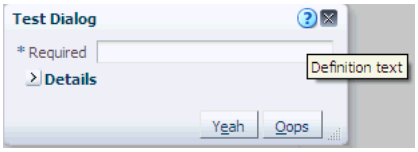
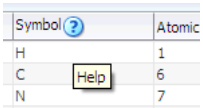


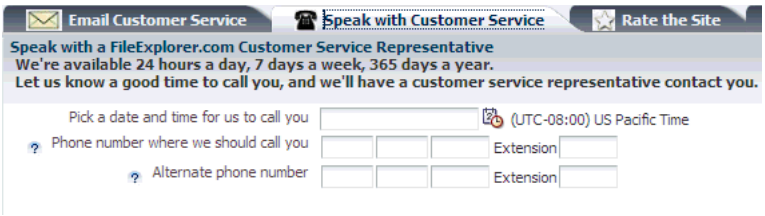
Table 17–1 shows the components that support Definition help.

Table 17–1 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	
Columns in table and tree	Below header text	

The second type of help is Instruction help. Where Instruction help is displayed depends on the component with which it is associated. The `panelHeader` and Search panel components display Instruction help within the header. Figure 17–18 shows how the text that typically is displayed as Definition help as shown in Figure 17–17 would be displayed as Instruction help within the `panelHeader` component.

Figure 17–18 Instruction Text for panelHeader

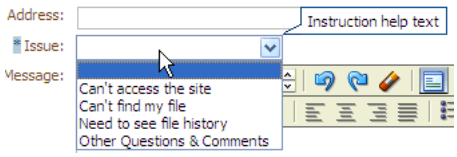


All other components that support Instruction help display the text within a note window, as shown in Figure 17–19. Note that no help icon is displayed.

Figure 17–19 Instruction Text for a Component

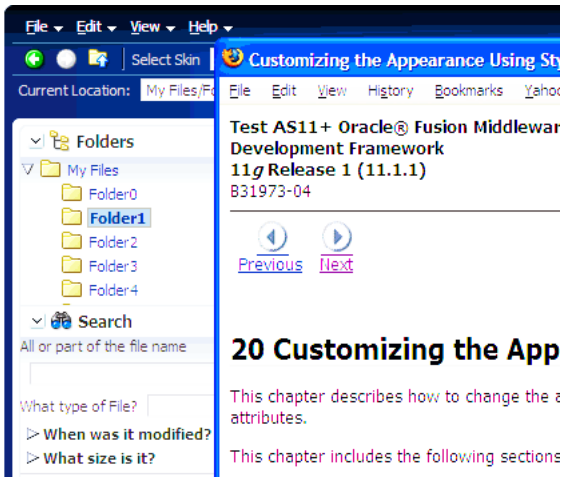
Table 17–2 shows the components that support Instruction help.

Table 17–2 Components That Support Instruction Help

Supported Components	Help Placement	Example
Input components, Choose Color, Choose Date, Quick Query	Note window, on focus only	
Select components	Note window, on hover and focus	
Panel Header, Panel Box, Query	Text below header text	

The last type of help is External URL help. You provide a URL to a web page in an external application, and when the help icon is clicked, the web page opens in a separate browser window, as shown in Figure 17–20. Instead of clicking a help icon, you can use JavaScript to open a help window based on any client-based event.

Figure 17–20 External URL Help



ADF Faces includes a variety of help providers. The `ResourceBundleHelpProvider` help provider allows you to create resource bundles that hold the help content. The `ELHelpProvider` help provider allows you to create XLIFF files that get converted into maps, or create a managed bean that contains a map of help text strings. You can use a combination of the different help providers. You can also create your own help provider class.

To create help for your application, do the following:

- Determine the help provider(s) to use and then implement the required artifacts.
- Register the help provider(s), specifying the prefix that will be used to access the provider's help. Each help provider has its own unique prefix, which is used as its identifier. A particular provider will be called to produce help only for help topic IDs that start with the prefix under which the provider is registered.
- 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 example, the value of the `helpTopicId` attribute on the `inputText` component shown in Figure 17–19 might be `RBHELP_FILE_NAME`, where `RBHELP` is the resource bundle help providers prefix, and `FILE_NAME` is the help topic name.

17.5.1 How to Create Resource Bundle-Based Help

You can store help text within standard resource bundle property files and use the `ResourceBundleHelpProvider` class to deliver the content.

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, `RBHELP`.
   - The topic name, for example, `TELEPHONE_NUMBER`.

- The help type, for example, `DEFINITION`.

For example, a topic ID might be `RBHELP_TELEPHONE_NUMBER_DEFINITION`.

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 (as shown in Example 17–2), then both types of help will be displayed by the UI component.

Example 17–2 shows an example resource bundle with three topics.

Example 17–2 Resource Bundle Help

```
RBHELP_CUST_SERVICE_EMAIL_DEFINITION=For security reasons,
                    we strongly discourage the submission of credit card numbers.
RBHELP_TELEPHONE_NUMBER_DEFINITION=We only support calling telephone numbers
                                                    in the United States at this time.
RBHELP_TELEPHONE_NUMBER_INSTRUCTIONS=Enter a telephone number.
```

Note: If you wish to use an external URL help type, create a subclass of the `ResourceBundleHelpProvider` class. For more information, see Step 3.

---

2. Register the resource bundle as a help provider in the `adf-settings.xml` file (for information on creating the `adf-settings.xml` file if one does not exist, see Section A.5.1, "How to Configure for ADF Faces in adf-settings.xml").

   To register the provider, open the `adf-settings.xml` file, 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, and must match the prefix used in the resource bundle.

     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 and enter `oracle.adf.view.rich.help.ResourceBundleHelpProvider`.
   - `<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, for example, baseName.
- <value>: Create as a child element to the <property> element and 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.

Example 17–3 shows how the resource bundle in Example 17–2 would be registered in the adf-settings.xml file.

Example 17–3 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="RBHELP_">
    <help-provider-class>
      oracle.adf.view.rich.help.ResourceBundleHelpProvider
    </help-provider-class>
    <property>
      <property-name>baseName</property-name>
      <value>oracle.adfdemo.view.resource.DemoResources</value>
    </property>
  </help-provider>
</adf-faces-config>
</adf-settings>
```

3. If you want to use External URL help, then you also must extend the ResourceBundleHelpProvider class and implement the getExternalUrl method. Example 17–4 shows an example method.

Example 17–4 Overriding the getExternalURL Method

```
protected String getExternalUrl(FacesContext context, UIComponent component,
                                                                    String topicId)
  {
    if (topicId == null)
      return null;
    if (topicId.contains("TOPICID_ALL") ||
        topicId.contains("TOPICID_DEFN_URL") ||
        topicId.contains("TOPICID_INSTR_URL") ||
        topicId.contains("TOPICID_URL"))
      return http://www.myURL.com;
    else
      return null;
  }
```

In Example 17–4, all the topics in the method return the same URL. You would have to create separate if statements to return different URLs.

If you want the external window to be launched based on a component's client event instead of from a help icon, use a JavaScript function. For more information, see Section 17.5.4, "How to Use JavaScript to Launch an External Help Window."

17.5.2 How to Create XLIFF-Based Help

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.

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, `XLIFFHELP_CREDIT_CARD_DEFINITION`. In this example, `XLIFFHELP` will become the prefix used to access the XLIFF file. `CREDIT_CARD` is the topic name, and `DEFINITION` is the type of help.

     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 (as shown in Example 17–5), 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 is used to hold translated text.
   - `<note>`: Create as a direct child of the `<trans-unit>` element and enter a description for the help text.

   Example 17–5 shows an example of an XLIFF file that contains two topics.

Example 17–5 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="XLIFF_CREDIT_CARD_DEFINITION">
        <source>Credit Card Definition</source>
        <target/>
        <note>Credit Card definition text.</note>
      </trans-unit>
      <trans-unit id="XLIFF_CREDIT_CARD_INSTRUCTIONS">
        <source>Credit Card Instructions</source>
        <target/>
        <note>Credit card instruction text.</note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

2. Register XLIFF as a help provider in the `adf-settings.xml` file (for information on creating the `adf-settings.xml` file if one does not exist, see Section A.5.1, "How to Configure for ADF Faces in adf-settings.xml").

To register the provider, 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, and must match the prefix used in the XLIFF file.

  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 and enter `oracle.adf.view.rich.help.ELHelpProvider`.
- `<property>`: Create as a child element to the `<help-provider>` element. The property values define the actual help source.
- `<property-name>`: Create as a child element to the `<property>` element and enter a name for the help, for example, `helpSource`.
- `<value>`: Create as a child element to the `<property>` element and enter an EL expression that resolves to the XLIFF file, wrapped in the `adfBundle` EL function, for example, `#{adfBundle['project1xliff.view.Project1XliffBundle']}`.

Example 17–6 shows how the XLIFF file in Example 17–5 would be registered in the `adf-settings.xml` file.

Example 17–6 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="XLIFF">
    <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>
```

17.5.3 How to Create Managed Bean Help

To implement managed bean help, 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.

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 17–7.

Example 17–7 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("MAPHELP_CREDIT_CARD_DEFINITION",
                                          "Map value for credit card definition");
      _HELP_MAP.put("MAPHELP_CREDIT_CARD_INSTRUCTIONS",
                                          "Map value for credit card instructions");
      _HELP_MAP.put("MAPHELP_SHOPPING_DEFINITION",
                                          "Map value for shopping definition");
      _HELP_MAP.put("MAPHELP_SHOPPING_INSTRUCTIONS",
                                          "Map value for shopping instructions");
    }

}
```

The first string must contain the prefix, the topic name, and the help type, `for` example, `MAPHELP_CREDIT_CARD_DEFINITION`. In this example, MAPHELP will become the prefix used to access the bean. `CREDIT_CARD` is the topic name, and D`EFINITION` is the type of help. The second string is the help text.

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 (as shown in Example 17–7), 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 Step 4.

---

2. Register the managed bean in the `faces-config.xml` file. Example 17–8 shows the bean shown in Example 17–7 registered in the `faces-config.xml` file.

Example 17–8 Managed Bean Registration in the faces-config.xml File.

```
<managed-bean>
  <managed-bean-name>helpTranslationMap</managed-bean-name>
    <managed-bean-class>
      oracle.adfdemo.view.webapp.ELHelpProviderMapDemo
    </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

For more information about using and registering managed beans, see Section 2.6, "Creating and Using Managed Beans."

3. Register the managed bean as a help provider in the `adf-settings.xml` file (for information on creating the `adf-settings.xml` file if one does not exist, see Section A.5.1, "How to Configure for ADF Faces in adf-settings.xml").

   To register the provider, open the `adf-settings.xml` file and add the following elements:

   - `<help-provider>`: Create and 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 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. The property defines the map of help strings on the managed bean.
   - `<property-name>`: Create as a child element to the `<property>` element and enter a property name, for example `helpSource`.
   - `<value>`: Create as a child element to the `<property>` element and enter an EL expression that resolves to the help map on the managed bean.

   Example 17–9 shows how the bean in Example 17–8 would be registered in the `adf-settings.xml` file.

Example 17–9 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="MAPHELP_">
   <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>
```

4. If you want to use External URL help with the managed bean provider, then extend the `ELHelpProvider` class and implement the `getExternalUrl` method. Example 17–10 shows an example method.

Example 17–10 Overriding the getExternalURL Method

```
protected String getExternalUrl(FacesContext context, UIComponent component,
                                                                  String topicId)
  {
    if (topicId == null)
      return null;
    if (topicId.contains("TOPICID_ALL") ||
        topicId.contains("TOPICID_DEFN_URL") ||
        topicId.contains("TOPICID_INSTR_URL") ||
        topicId.contains("TOPICID_URL"))
      return http://www.myURL.com;
    else
      return null;
  }
```

In Example 17–10, all the topics in the method return the same URL. You must create separate `if` statements to return different URLs.

If you want the external window to be launched based on a component's client event instead of from a help icon, use a JavaScript function. For more information, see Section 17.5.4, "How to Use JavaScript to Launch an External Help Window."

17.5.4 How to Use JavaScript to Launch an External Help Window

If you want to use 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.

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 17–11 shows the `launchHelp` function used to launch the `helpClient.jspx`.

Example 17–11 JavaScript to Launch an External Help Page

```
<af:resource type="javascript">
  function launchHelp(event)
    {
      AdfPage.PAGE.launchHelpWindow("helpClient.jspx");
    }
</af:resource>
```

2. Drag and drop a component whose client event will cause the function to be called. You must set the `clientId` on this component to `true`.
3. In the Component Palette, 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 3.2, "Listening for Client Events."

Example 17–12 shows the code used to have a click event on a `commandToolbarButton` component launch the `helpClient.jspx` page.

Example 17–12 Page Code Used to Launch an External Help Window

```
<af:toolbar id="tb1">
  <af:commandToolbarButton text="Launch help window" id="ctb1"
                           icon="/images/happy_computer.gif">
    <af:clientListener method="launchHelp" type="click"/>
  </af:commandToolbarButton>
</af:toolbar>
<af:resource type="javascript">
  function launchHelp(event)
    {
      AdfPage.PAGE.launchHelpWindow("helpClient.jspx");
    }
</af:resource>
```

17.5.5 How to Create a 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 ADF Faces Javadoc.

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.
3. 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, the ADF Faces providers all use a property to define the actual source of the help strings. To access a property in the `adf-settings.xml` file, create a method that sets a property that is a `String`. For example:

   ```
   public void setMyCustomProperty(String arg)
   ```

4. To register the provider, 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.

Example 17–13 shows an example of a help provider class registered in the `adf-settings.xml` file.

Example 17–13 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>
```

17.5.6 How to Access Help Content from a UI Component

Use the `HelpTopicId` attribute on components to access and display the help.

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 17–1 and Table 17–2.
2. In the Property Inspector, expand the **Appearance** section, and enter a value for the `helpTopicId` attribute. This 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="Credit Card" helpTopicId="XLIFF_CREDIT_CARD"/>
```

This example will return both the definition and instruction help defined in the XLIFF file in Example 17–5.

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['XLIFF_CREDIT_CARD'].instructions}"/>
```

This will access the instruction help text.

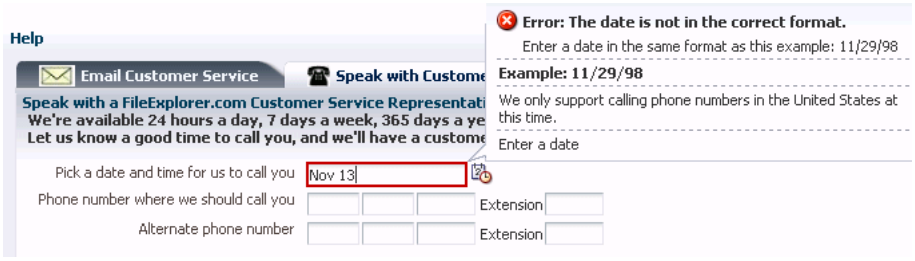
17.5.7 What You May Need to Know About Combining Different Message Types

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 17–21 shows an `inputDate` component that contains a converter, instruction help, and a tip message.

Figure 17–21 Different Message Types Can Be Displayed at One Time



18

Working with Navigation Components

This chapter describes how to use ADF Faces navigation components such as `commandButton`, `navigationPane`, and `train` to provide navigation in web user interfaces.

This chapter includes the following sections:

18.1 Introduction to 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 (or other 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 4, "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 Section 7.4, "Using Partial Page Navigation".

Command components in ADF Faces include:

- Button and link components for navigating to another location with or without server-side actions. See Section 18.2, "Using Buttons and Links for Navigation".

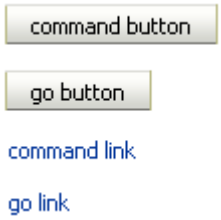
- Components that render items such as tabs and breadcrumbs for navigating hierarchical pages. See Section 18.5, "Using Navigation Items for a Page Hierarchy".
- Train components for navigating a multistep process. See Section 18.8, "Using Train Components to Create Navigation Items for a Multi-Step Process".

In addition to using command components for navigation, ADF Faces also includes listener tags that you can use in conjunction with command components to have specific functionality execute when the action event fires. For more information, see Section 18.4, "Using Buttons or Links to Invoke Functionality".

18.2 Using Buttons and Links for Navigation

Buttons and links in ADF Faces include the command components `commandButton`, `commandLink`, and `commandImageLink`, as well as the go components `goButton`, `goImageLink`, and `goLink`. The main difference between command components and go components is that while command components submit requests and fire `ActionEvent` events, go components navigate directly to another location without delivering an action. Visually, the rendered command and go components look the same, as shown in Figure 18–16.

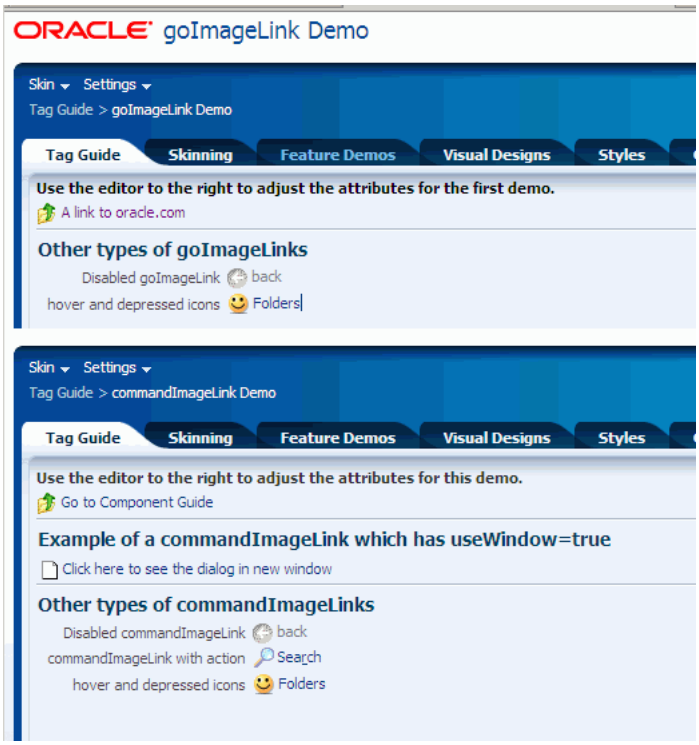
Figure 18–1 Command Components and Go Components



> **Tip:** ADF Faces also provides specialized command components that can be used inside menus and toolbars. For more information, see Chapter 14, "Using Menus, Toolbars, and Toolboxes".

The `commandImageLink` and `goImageLink` components render images as links, along with optional text, as shown in Figure 18–2. 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.

Figure 18–2 Command Image Link and Go Image Link



ADF Faces also includes a toolbar button that provides additional functionality, such as a `popup` facet that can open popup menus from a toolbar button. For more information, see Section 14.3, "Using Toolbars".

You can configure your application to allow end users invoke a browser's context menu when they right-click a command component that renders a link. End users who right-click the link rendered by a command 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 18.3, "Configuring a Browser's Context Menu for Command Links."

You can show a warning message to users if the page that they attempt to navigate away from contains uncommitted data. Add the `checkUncommittedDataBehavior` component as a child to command 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 following components:

- `af:commandButton`
- `af:commandLink`
- `af:commandImageLink`
- `af:commandToolbarButton`
- `af:activeCommandToolbarButton`

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 8.2.5, "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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

18.2.1 How to Use Command Buttons and Command Links

Typically, you use `commandButton`, `commandLink`, and `commandImageLink` components to perform page navigation and to execute any server-side processing.

To create and use command components:

1. Create a `commandButton` component by dragging and dropping a **Button** from the Component Palette to the JSF page. Create a `commandLink` component by dragging and dropping a **Link**. Create a `commandImageLink` component by dragging and dropping an **Image Link**.
2. In the Property Inspector, expand the **Common** section and set the `text` attribute.

   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 22.3.4, "How to Define Access Keys for an ADF Faces Component"

3. Set the `icon` attribute to the URI of the image file you want to use for inside a `commandButton` or `commandImageLink` component (this is not supported for `commandLink`). For a `commandImageLink` component, you can also set the `hoverIcon`, `disabledIcon`, and `depressedIcon` attributes.

   Tip: You can use either the `text` attribute (or `textAndAccessKey` attribute) or the `icon` attribute, or both.

4. Set the `action` attribute 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 2.3, "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 13, "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://download.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 is used to perform 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.

5. Expand the **Behavior** section and set the `disabled` attribute to `true` if you want to show the component as a non-interactive button or link.
6. Set the `partialSubmit` attribute to `true` to fire a partial page request each time the component is activated. For more information, see Section 7.2, "Enabling Partial Page Rendering Declaratively".
7. Set the `immediate` attribute to `true` if you want 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 4.2, "Using the Immediate Attribute".
8. Optionally, if you set the `immediate` attribute to `true` as described in step 7, you can add the `af:checkUncommittedDataBehavior` component as a child to the command component to display a warning message to the user if the page contains uncommitted data. Drag the **Check Uncommitted Data Behavior** from the Behavior section of the Operations panel in the Component Palette and drop it as a child of the command component you added in step 1.

Note: You must have also set the `document` tag's `uncommittedDataWarning` attribute to `on`, as described in Section 8.2.5, "How to Configure the document Tag."

Command 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 Chapter 18, "Working with Navigation Components".

18.2.2 How to Use Go Buttons and Go Links

You use the `goButton`, `goImageLink`, and `goLink` components to perform direct page navigation, without delivering an `ActionEvent` event.

To create and use go buttons and go links:

1. Create a `goButton` component by dragging and dropping a **Go Button** from the Component Palette to the JSF page. Create a `goLink` component by dragging and dropping a **Go Link**. Create a `goImageLink` component by dragging and dropping a **Go Image Link**.

2. In the Property Inspector, expand the **Common** section and set the `text` attribute if you created a `goButton` or `goLink` component. If you created a `goImageLink` component, you set the `text` attribute in the **Other** section.

   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 22.3.4, "How to Define Access Keys for an ADF Faces Component"

3. Set the `icon` attribute to the URI of the image file you want to use for inside a `goButton` or `goImageLink` component (not supported for `goLink`). For a `goImageLink` component, you can also set the `hoverIcon`, `disabledIcon`, `depressedIcon`, and `iconPosition` attributes.

   The `iconPosition` attribute supports two values: `leading` (default) and `trailing`. Set to `leading` to render the icon before the text. Set to `trailing` to render the icon after the text.

   Tip: You can use either the `text` attribute (or `textAndAccessKey` attribute) or the `icon` attribute, or both.

4. Set the `destination` attribute to the URI of the page to which the link should navigate.

   For example, in the File Explorer application, the `goLink` component in the `popups.jspx` file has the following EL expression set for its `destination` attribute:

   ```
   destination="http://www.oracle.com"
   ```

5. Set the `targetFrame` attribute to specify where the new page should display. Acceptable values are:

   - `_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.

6. Expand the **Behavior** section and set the `disabled` attribute to `true` if you want to show the component as a non-interactive button or link. You set the `disabled` attribute for the `goImageLink` component in the **Other** section.

18.3 Configuring a Browser's Context Menu for Command Links

The command 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 command components. 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. The components for which you can configure this behavior include the following:

- af:commandLink
- af:commandImageLink
- af:commandMenuItem (stand-alone or 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. Examples of these components include the following:

- af:goLink
- af:goImageLink
- af:commandNavigationItem where you specify a value for the destination attribute and no value for the action attribute

18.3.1 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 command components you can configure this functionality for. For more information, Section 18.3, "Configuring a Browser's Context Menu for Command Links."

To configure a browser's context menu for a command link:

1. In the Application Navigator, double-click **web.xml** to open the file.

   By default, JDeveloper opens the web.xml file in the Overview editor.

2. In the Context Initialization Parameters table, 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.

18.3.2 What Happens When You Configure a Browser's Context Menu for Command Links

If you followed the procedure outlined in Section 18.3, "Configuring a Browser's Context Menu for Command Links," JDeveloper writes a value to the web.xml file, as shown in Example 18–1.

Example 18–1 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 18.3, "Configuring a Browser's Context Menu for Command Links."

18.4 Using Buttons or Links to Invoke Functionality

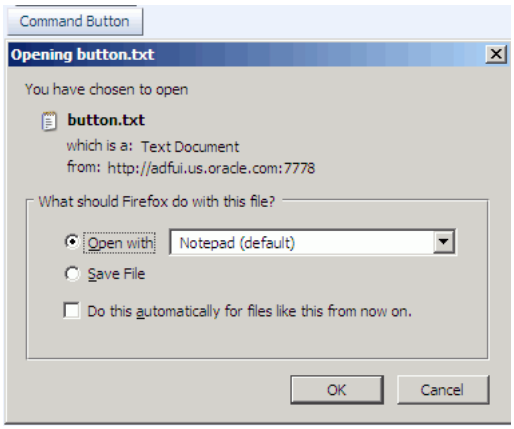
In addition to using command components for navigation, ADF Faces also includes listener tags that you can use in conjunction with command components to have specific functionality execute when the action event fires. Listener tags included with ADF Faces include:

- `exportCollectionActionListener`: Use to export data from an ADF Faces application to an Excel spreadsheet. For more information, see Section 10.9, "Exporting Data from Table, Tree, or Tree Table".
- `fileDownloadActionListener`: Use to initiate a file download from the server to the local computer. For more information, see Section 18.4.1, "How to Use a Command Component to Download Files".
- `resetActionListener`: Use to reset submitted values. However, no data model states will be altered. For more information, see Section 18.4.2, "How to Use a Command Component to Reset Input Fields". 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 9.2.3, "How to Add a Reset Button to a Form".

18.4.1 How to Use a Command Component to Download Files

You can create a way for users to download files by creating an action component such as a command button and associating it with a `fileDownloadActionListener` tag. When the user selects or clicks the action component, a popup dialog is displayed that allows the user to select different download options, as shown in Figure 18–3.

Figure 18–3 File Download Dialog



The `fileDownloadActionListener` tag is used declaratively to allow an action component such as command button, command link, or menu item to programmatically send the contents of a file to the user. You can also declare a specific content type or file name. Because file download must be processed with an ordinary request instead of the `XMLHttp AJAX` requests, the parent component's `partialSubmit` attribute, if supported, must be set to `false`.

> **Tip:** For information about uploading a file to the server, see Section 9.9, "Using File Upload".

After the content has been sent to the browser, how that content is displayed or saved depends on the option selected in the dialog. If the **Open with** option was selected, 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 **Save to Disk** option was selected, depending on the browser, a popup dialog may appear to select a file name and a location in which to store the content.

Example 18–2 shows the tags of a command button with the `fileDownloadActionListener` tag to download the file content `Hi there!` to the user.

Example 18–2 File Download Using Command Button and fileDownloadActionListener Tag

```
<af:commandButton value="Say Hello">
  <af:fileDownloadActionListener filename="hello.txt"
      contentType="text/plain; charset=utf-8"
      method="#{bean.sayHello}"/>
</af:commandButton>
```

Example 18–3 shows the managed bean method used to process the file download.

Example 18–3 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!");
   . . .
}
```

To create a file download mechanism:

1. From the Component Palette, drag and drop any action component to your page (for more information about action components, see Section 18.2, "Using Buttons and Links for Navigation").
2. Expand the **Operations** section of the Component Palette, and drag and drop the **File Download Action Listener** tag as a child to the action component.
3. In the Property Inspector 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 be used to 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, the code for a command button would be similar to the following:

```
<af:commandButton text="Load File">
  <af:fileDownloadActionListener contentType="text/plain"
      filename="MyFile.txt"
      method="#(mybean.LoadMyFile"/>
</af:commandButton>
```

18.4.2 How to Use a Command Component to Reset Input Fields

You can use the `resetActionListener` tag in conjunction with a command component to reset input values. All values will be returned 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 9.2.3, "How to Add a Reset Button to a Form".

To use the reset tag:

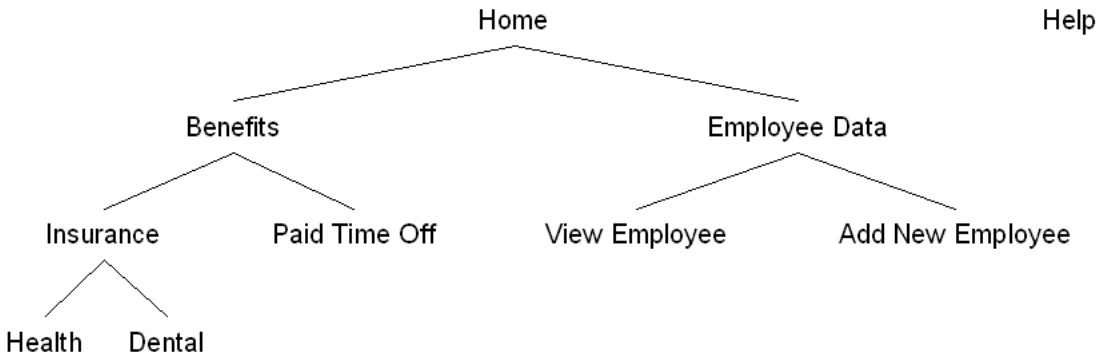
1. Create a command component as documented in Section 18.2, "Using Buttons and Links for Navigation".
2. Drag and drop a **Reset Action Listener** from the Component Palette as a child to the command component.

18.5 Using Navigation Items for a Page Hierarchy

Note: If your application uses the Fusion technology stack with the 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" section of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 18–4 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 18–4 Benefits and Employee Page Hierarchy



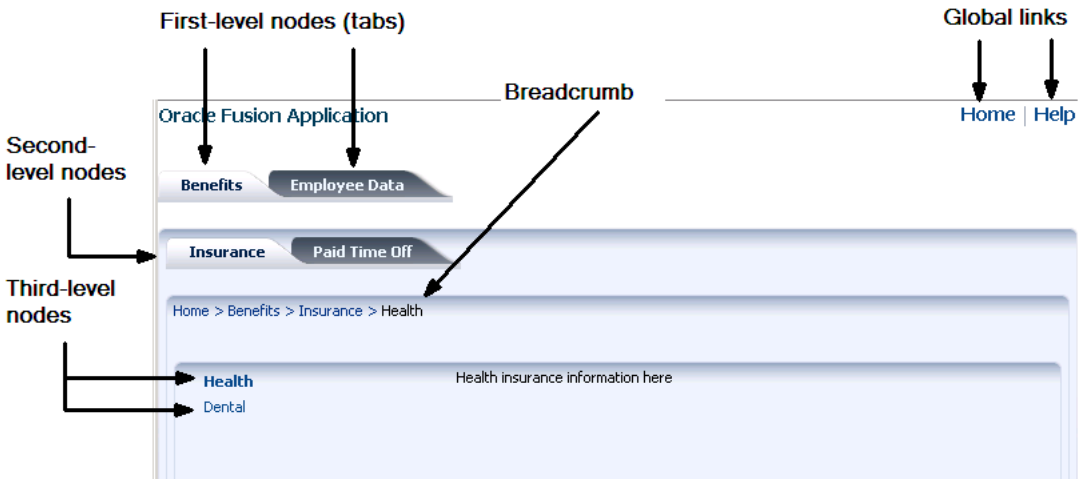
Navigation in a page hierarchy follows the parent-child links. For example, to view Health information, the user would start drilling from the Benefits page, then move to the Insurance page where two choices are presented, one of which is Health. The path of selected links starting from Home and ending 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 18–4, 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 a page hierarchy are tabs, bars, lists, and global links, all of which can be created by using the `navigationPane` component. Figure 18–5 shows the hierarchy illustrated in Figure 18–4, as rendered using the `navigationPane` and other components.

Figure 18–5 Rendered Benefits and Employee Pages



In general, tabs are used as first-level nodes, as shown in Figure 18–5, where there are tabs for the Benefits and Employee Detail 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 use tabs for both first- and second-level nodes. Global links (which represent global nodes) are rendered as text links. In Figure 18–5, the Home and Help global links are rendered as text links.

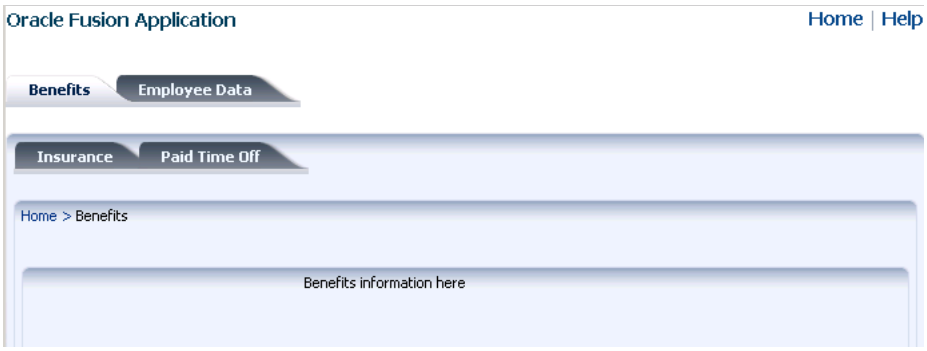
One `navigationPane` component corresponds to one level of nodes, whether they are first-, second-, or third-level nodes, or global nodes. Regardless of the type of navigation items the `navigationPane` component is configured to render for a level, you always use the `commandNavigationItem` component to represent each item within the `navigationPane` component.

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 18–10 and Figure 18–11, 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 20, "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 19.3, "Using Page Templates".

For each page of simpler 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 links for each level. For example, to create the Health insurance page as shown in Figure 18–5, 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. If instead you were creating the Benefits page, as shown in Figure 18–6, you would create only three `navigationPane` components (one each for the global, first, and second levels), and then create just the `commandNavigationItem` components for the links seen from this page.

Figure 18–6 First-Level Page



As you can see, with large 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. 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 JSF navigation model, through the default `ActionListener` mechanism, is used to choose the page to navigate to when users select a navigation item. For more information about the menu model, see Section 18.6, "Using a Menu Model to Create a Page Hierarchy".

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 18.6, "Using a Menu Model to Create a Page Hierarchy". For more information about manually creating a navigational hierarchy, see Section 18.7, "Creating a Simple Navigational 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 14, "Using Menus, Toolbars, and Toolboxes".

18.6 Using a Menu Model to Create a Page Hierarchy

> **Note:** If your application uses the Fusion technology stack or the 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" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Section 18.5, "Using Navigation Items for a Page Hierarchy" describes how you can create a navigation menu 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 menu 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 10.5, "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 `XMLMenuModel` class to create the menu model.

> **Performance Tip:** Using the `navigationPane` component with the menu model results in a full page refresh every time the user switches the tab. Instead, you can use the `panelTabbed` component (see Section 8.9, "Displaying or Hiding Contents in Accordion Panels and Tabbed Panels". This component has built-in support for partial page rendering of the tabbed content. However, it 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 and then create the `XMLMenuModel` metadata. See Section 18.6.1, "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 18.6.2, "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 19, "Creating and Reusing Fragments, Page Templates, and Components".

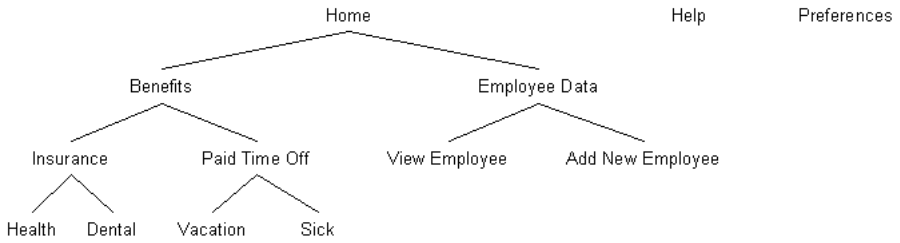
- On each page, bind the `navigationPane` and `breadCrumbs` components to the `XMLMenuModel` class. See Section 18.6.3, "How to Bind to the XMLMenuModel in the JSF Page" and Section 18.6.4, "How to Use the breadCrumbs Component".

18.6.1 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. In the `XMLMenuModel` metadata file, the entire 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.

The remaining nodes in the 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 as depicted in Figure 18–7.

Figure 18–7 Sample Page Hierarchy



If you wanted each node in the hierarchy to have its own page to which a user can navigate, then you would create an item node in the metadata for each page. You nest the children nodes inside the 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.

You can also nest menu models using the shared nodes. This approach is recommended 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 model so that it could be reused across an application. Instead of creating the nodes for each use, you could instead create the

nodes once as a separate menu and then within the different hierarchies, use a shared node to reference the Benefits menu model.

Example 18–4 shows an `XMLMenuModel` metadata file for defining a page hierarchy illustrated in Figure 18–7.

Example 18–4 XMLMenuModel Metadata File Sample

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <itemNode id="Home" focusViewId="/home.jspx" label="Home" action="goHome">
    <itemNode id="benefits" focusViewId="/benefits.jspx" action="goBene"
              label="Benefits">
      <itemNode id="insurance" focusViewId="/insurance.jspx" action="goIns"
                label="Insurance">
        <itemNode id="health" focusViewId="/health.jspx" action="goHealth"
                  label="Health"/>
        <itemNode id="dental" focusViewId="/dental.jspx" action="goDental"
                  label="Dental"/>
      </itemNode>
      <itemNode id="pto" focusViewId="/pto.jspx" action="goPto"
                label="Paid Time Off">
        <itemNode id="vacation" focusViewId="/vacation.jspx"
                  action="goVacation" label="Vacation"/>
        <itemNode id="sick" focusViewId="/sick.jspx" action="goSick"
                  label="Sick Pay"/>
      </itemNode>
    </itemNode>
    <groupNode id="empData" idref="newEmp" label="Employee Data">
      <itemNode id="newEmp" focusViewId="/createemp.jspx" action="goCreate"
                label="Create New Employee"/>
      <itemNode id="viewdata" focusViewId="/viewdata.jspx" action="goView"
                label="View Data"/>
    </groupNode>
  </itemNode>
  <itemNode id="Help" focusViewId="/globalhelp.jspx" action="goHelp"
            label="Help"/>
  <itemNode id="Preferences" focusViewId="/preferences.jspx" action="goPref"
            label="Preferences"/>
</menu>
```

Within the root `menu` element, global nodes are any types of nodes that are direct children of the `menu` element; in other words, the first level of elements under the `menu` element are global nodes. For example, the code in Example 18–4 shows three global nodes, namely, Home, Help, and Preferences. Within a first-level child node, nodes can be nested to provide more levels of navigation. For example, the code in Example 18–4 shows two second-level nodes under Home, namely, Benefits and Employee Data. Within Benefits, there are two third-level nodes, Insurance and Paid Time Off, and so on.

JDeveloper simplifies creating metadata for an `XMLMenuModel` class by providing the Create ADF Menu Model wizard.

To create the XMLMenuModel metadata:

1. 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 18–4 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 18–5.

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 18–5 Global Navigation Rule for a Page Hierarchy in faces-config.xml

```
<navigation-rule>
  <navigation-case>
    <from-outcome>goHome</from-outcome>
    <to-view-id>/home.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHelp</from-outcome>
    <to-view-id>/globalhelp.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goEmp</from-outcome>
    <to-view-id>/empdata.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goBene</from-outcome>
    <to-view-id>/benefits.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goIns</from-outcome>
    <to-view-id>/insurance.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goPto</from-outcome>
    <to-view-id>/pto.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goView</from-outcome>
    <to-view-id>/viewdata.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goCreate</from-outcome>
    <to-view-id>/createemp.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHealth</from-outcome>
    <to-view-id>/health.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goDental</from-outcome>
    <to-view-id>/dental.jspx</to-view-id>
  </navigation-case>
...
</navigation-rule>
```

For more information about creating navigation cases in JDeveloper, see Section 2.3, "Defining Page Flows".

2. In the Application Navigator, locate the project where you wish to create the `XMLMenuModel` metadata file. Under the project's **Web Content - WEB-INF** folder, right-click the `faces-config.xml` file, and choose **Create ADF Menu** from the context menu.

> **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" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

3. In the Create ADF Menu Model dialog, enter a file name for the `XMLMenuModel` metadata file, for example, `root_menu`.
4. Enter a directory for the metadata file. By default, JDeveloper will save the `XMLMenuModel` metadata file in the `WEB-INF` directory of the application.

   When you click **OK**, JDeveloper automatically does the following for you:

   - Creates a managed bean for the model in the `faces-config.xml` file, using the name specified in Step 3 for the managed bean name.
   - Sets the value of the managed bean's `source` managed property to the `XMLMenuModel` metadata file, specified in Step 3, for example, `/WEB-INF/root_menu.xml`.
   - Displays the source file (that is, `/WEB-INF/root_menu.xml`) as a blank `XMLMenuModel` metadata file in the source editor, as shown in Example 18–6.

Example 18–6 Blank XMLMenuModel Metadata File

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"></menu>
```

   For more information about the managed bean configuration JDeveloper automatically adds for you, see Section 18.6.2, "What Happens When You Use the Create ADF Menu Model Wizard".

5. Select the **menu** node in the Structure window and enter the appropriate information in the Property Inspector.

   Table 18–1 shows the attributes you can specify for the `menu` element.

Table 18–1 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 18–7 for a sample `XMLMenuModel` metadata file that uses a resource bundle.
`xmlns`	Required. Set to `http://myfaces.apache.org/trinidad/menu`

   Example 18–7 shows sample `XMLMenuModel` metadata code that uses EL expressions to access a resource bundle for the navigation item labels.

Example 18–7 XMLMenuModel Using Resource Bundle

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"
```

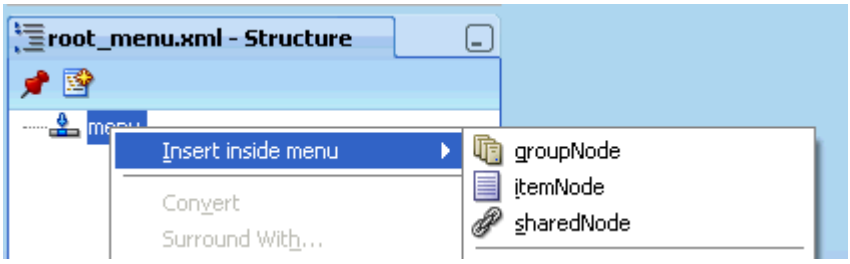
```
      resourceBundle="org.apache.myfaces.demo.xmlmenuDemo.resource.MenuBundle"
      var="bundle">
  <itemNode id="in1" label="#{bundle.somelabel1}" ../>
  <itemNode id="in2" label="#{bundle.somelabel2}" ../>
</menu>
```

Note: When you use a `sharedNode` element to create a submenu and you use resource bundles for the navigation item labels, it is quite 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 21, "Internationalizing and Localizing Pages".

6. In the Structure window, add the desired elements for the nodes in your hierarchy, using `itemNode`, `groupNode`, or `sharedNode` as needed. To begin, right-click **menu** and choose **Insert inside menu**, and then choose the desired element from the context menu, as shown in Figure 18–8.

Figure 18–8 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.

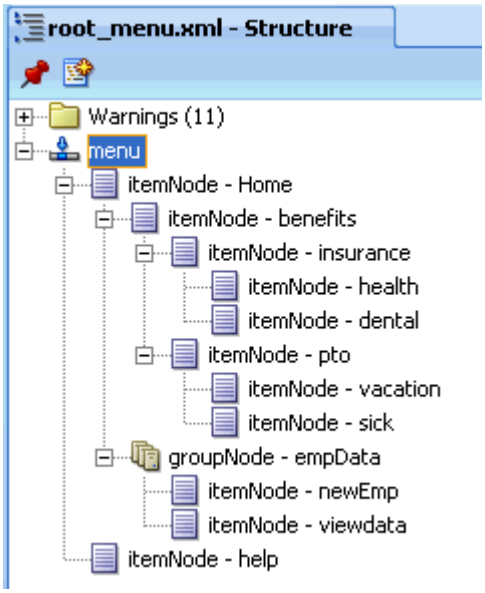
  You can insert a `sharedNode` element anywhere within the hierarchy. For example, in the code shown in Example 18–8, the `sharedNode` element adds a submenu on the same level as the global nodes.

Example 18–8 SharedNode Sample Code

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu"
  <itemNode id="in1" label="Home" ../>
  <sharedNode ref="#{shared_menu}"/>
  <itemNode id="in6" 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 18–9.

Figure 18–9 Tree Structure of XMLMenuModel Metadata in Structure Window



7. For each element used to create a node, set the properties in the Property Inspector, as described in Table 18–2 for `itemNode` elements, Table 18–3 for `groupNode` elements, and Table 18–4 for `sharedNode` elements.

Table 18–2 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`.

Table 18–2 (Cont.) itemNode Element Attributes

Attribute	Description
`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.jspx` (as configured in the `faces-config.xml` file), then `focusViewId` must also be `/page_one.jspx`. 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 18–4, 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.
`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 18–9. 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 18–3 shows the attributes you must specify when you use a `groupNode` element.

Example 18–9 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.jspx"/>
      <itemNode id="in2" label="LEVEL2_TAB_1" action="goToSubTabTwo"
                focusViewId="/menuDemo/subtab2.jspx"/>
    </groupNode>
    <itemNode id="in3" label="PRIMARY_TAB_1" focusViewId="/menuDemo/tab2.jspx"
              destination="/faces/menuDemo/tab2.jspx"/>
  </groupNode>
  <itemNode id="gin1" label="GLOBAL_TAB_1" action="goToGlobalOne"
            focusViewId="/menuDemo/global1.jspx"/>
  <itemNode id="gin2" label="GLOBAL_TAB_2"
            destination="/faces/menuDemo/global2.jspx"
            focusViewId="/menuDemo/global2.jspx"/>
</menu>
```

Table 18–3 GroupNode Element Attribute

Attribute	Description
`id`	A unique identifier for the group node. As shown in Example 18–4, it is good practice to use `gnX` for the ID of each `groupNode`, where for example, `gnX` could be `gn1`, `gn2`, and so on.
`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 18–4 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.

18.6.2 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 metadata file in the `faces-config.xml` file, using the metadata file name you provide as the managed bean name.

Example 18–10 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 18–10 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 each `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 source menu metadata 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.

  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 this value already set to properly parse and create the menu's XML metadata from the `source` managed property.

- `source`: Specifies the source metadata file to use.

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 18–8), 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 18–11.

Example 18–11 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>true</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 file, 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.

18.6.3 How to Bind to the XMLMenuModel in the JSF Page

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 18–4), 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 18–7 and Example 18–4, 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 18.7.1, "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 `XMLMenuModel` managed bean, and insert only one `commandNavigationItem` component into the `nodeStamp` facet of each `navigationPane` component, as shown in Example 18–12.

Example 18–12 navigationPane Component Bound to XMLMenuModel Managed Bean

```
<af:navigationPane var="menuNode" value="#{root_menu}" level="1"
                   hint="tabs">
  <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` code for that level.

To bind to the XMLMenuModel managed bean:

1. If you want the menu tabs to be styled, create a `decorativeBox` component by dragging and dropping a **Decorative Box** from the **Layout** section of the Component Palette to the JSF page. Set the theme to determine how you want the tabs to appear. Valid values are:

   - `default`: Body is white with a blue border. Top-left corner is rounded.
   - `light`: Body is light blue. Top-left corner is rounded.
   - `medium`: Body is medium blue. Top-left corner is rounded.
   - `dark`: Body is dark blue. Top-left corner is rounded.

   You can change how the themes are displayed. For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins".

2. Create a `navigationPane` component by dragging and dropping a **Navigation Pane** from the Component Palette to the JSF page. Add a `navigationPane` component for each level of the hierarchy.

   Tip: The Navigation Pane component can be found in the **Layout** pane of the Component Palette.

   For example, to create any of the pages as shown in the hierarchy in Figure 18–5, you would drag and drop four `navigationPane` components.

3. For each `navigationPane` component, in the Property Inspector, expand the **Common** section and set the `Hint` attribute to one of the following types of navigation items to determine how the `navigationPane` will display the following:

   - bar: Displays the navigation items separated by a bar, for example the **Insurance** and **Paid Time Off** links in Figure 18–11.
   - buttons: Displays the navigation items separated by a bar in a global area, for example the **Home** and **Help** links in Figure 18–11.
   - 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 18–11.
   - tabs: Displays the navigation items as tabs, for example the **Benefits** and **Employee Data** tabs in Figure 18–11.

4. Set the `level` attribute to point to 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 18–4), 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.

5. In the Property Inspector, expand the **Data** section. Set the `value` attribute 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.

6. Set the `var` attribute 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.

7. Drag and drop a **Navigation Item** from the Component Palette to the `nodeStamp` facet of the `navigationPane` component.

8. 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 6 along with the name of the corresponding `itemNode` element that holds the value in the metadata. Table 18–5 shows the attributes on the navigation item that has corresponding values in the metadata.

Table 18–5 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 18–13 shows the JSF code for binding to a menu model for the HR example.

Example 18–13 Binding to the XML Model

```
<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>
```

18.6.4 How to Use the breadCrumbs Component

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.

To create a breadcrumb:

1. Create a `breadCrumbs` component by dragging and dropping a **Bread Crumbs** component from the Component Palette to the JSF page.
2. By default, breadcrumb links are displayed in a horizontal line. To change the layout to be vertical, in the Property Inspector, expand the **Common** section and set the `orientation` attribute to `vertical`.
3. In the Property Inspector, expand the **Data** section. Set the `value` attribute to the root menu model 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 18–8), 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.

4. Set the `var` attribute 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.

5. Add one `commandNavigationItem` component as a child by dragging and dropping a **Navigation Item** from the Component Palette to 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.

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 `breadCrumbs` component in Step 4 along with the name of the corresponding `itemNode` element that holds the value in the metadata. Table 18–5 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 18–14 breadCrumbs Component Bound to a MenuModel

```
<af:breadCrumbs var="menuNode" value="#{root_menu}">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuNode.label}"
                              action="#{menuNode.doAction}"/>
  </f:facet>
</af:breadCrumbs>
```

18.6.5 What Happens at Runtime

The `value` attribute of `navigationPane` component 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 `navigationPane` 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 navigationPane 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.

18.6.6 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 18–15 shows an example of the required code.

Example 18–15 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;
}
```

18.7 Creating a Simple Navigational Hierarchy

Figure 18–10 and Figure 18–11 show an example of what the user interface looks like when the `navigationPane` component and individual `commandNavigationItem` components are used to create a view for the page hierarchy shown in Figure 18–4.

Figure 18–10 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 `navigationPanes` and `commandNavigationItem` components needed for each page, as well as to determine whether or not each component should be configured as selected when the user visits the page. For example, in Figure 18–10,

which shows the Employee Data page, only the child bars of Employee Data are needed, and the Employee Data tab renders as selected.

Similarly in Figure 18–11, which shows the Health page, only the child bars of Benefits are needed, and the Benefits tab must be configured as selected. Additionally for this page, you would create the child nodes under Insurance, which can be presented as vertical lists on the side of the page. The contents of the page are displayed in the middle, to the right of the vertical lists.

Figure 18–11 Navigation Items Available from the Health Page

Regardless of the type of navigation items you use (such as tabs or bars), a series of `commandNavigationItem` child components within each `navigationPane` component provide the actual navigation items. For example, in Figure 18–11 the actual link for the Employee Data tab, the Insurance and Paid Time Off bars, and the Health and Dental links in the list are each provided by a `commandNavigationItem` component.

18.7.1 How to Create a Simple Page Hierarchy

When your navigational hierarchy contains only a few pages and is not very deep, you can elect to manually create the hierarchy. Doing so involves creating the navigation metadata, using the `navigationPane` component to create the hierarchy, and using the `commandNavigationItem` component to create the links.

To manually create a navigation hierarchy:

1. 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 18–4 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 18–16.

   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 18–16 Global Navigation Rule for a Page Hierarchy in faces-config.xml

```
<navigation-rule>
  <navigation-case>
    <from-outcome>goHome</from-outcome>
    <to-view-id>/home.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHelp</from-outcome>
```

```
    <to-view-id>/globalhelp.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goEmp</from-outcome>
    <to-view-id>/empdata.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goBene</from-outcome>
    <to-view-id>/benefits.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goIns</from-outcome>
    <to-view-id>/insurance.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goPto</from-outcome>
    <to-view-id>/pto.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goView</from-outcome>
    <to-view-id>/viewdata.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goCreate</from-outcome>
    <to-view-id>/createemp.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goHealth</from-outcome>
    <to-view-id>/health.jspx</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>goDental</from-outcome>
    <to-view-id>/dental.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

For more information about creating navigation cases in JDeveloper, see Section 2.3, "Defining Page Flows".

2. If you want the menu tabs to be styled, create a `decorativeBox` component by dragging and dropping a **Decorative Box** from the **Layout** section of the Component Palette to the JSF page. Set the theme to determine how you want the tabs to appear. Valid values are:

   - `default`: Body is white with a blue border. Top-left corner is rounded.
   - `light`: Body is light blue. Top-left corner is rounded.
   - `medium`: Body is medium blue. Top-left corner is rounded.
   - `dark`: Body is dark blue. Top-left corner is rounded.

   You can change how the themes are displayed. For more information, see Chapter 20, "Customizing the Appearance Using Styles and Skins".

3. Create a `navigationPane` component by dragging and dropping a **Navigation Pane** from the **Layout** section of the Component Palette as a child to the `decorativeBox` component. Add a `navigationPane` component for each level of the hierarchy.

   For example, to create the Health page as shown in Figure 18–11, 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.

4. For each `navigationPane` component, in the Property Inspector, expand the **Common** section and set the `Hint` attribute to one of the following types of navigation items to determine how the `navigationPane` component will be displayed:
   - bar: Displays the navigation items separated by a bar, for example the **Insurance** and **Paid Time Off** links in Figure 18–11.
   - buttons: Displays the navigation items separated by a bar in a global area, for example the **Home** and **Help** links in Figure 18–11.
   - 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 18–11.
   - tabs: Displays the navigation items as tabs, for example the **Benefits** and **Employee Data** tabs in Figure 18–11.
5. For each `navigationPane` component, add the needed `commandNavigationItem` components to represent the different links by dragging and dropping a **Navigation Item** from the **Common Components** section of the Component Palette. 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 18–11, you would use a total of six `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.

6. For each `commandNavigationItem` component, set the navigation to the desired page. In the Property Inspector, expand the **Common** section and provide a static string outcome of an action or use an EL expression to reference an action method through the `action` property. If you use a string, it must match the navigation metadata set up in the navigation rules for the page created in Step 1. If referencing a method, that method must return the required string.
7. In the Property Inspector, expand the **Behavior** section and set the `selected` attribute. This attribute should be `true` 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 and the appearance changes to indicate to

the user that the item has been selected. For example, in Figure 18–11 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 18–17 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 18–17 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>
```

18.7.2 How to Use the breadCrumbs Component

In both Figure 18–10 and Figure 18–11, 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 create such a path of links, you use the `breadCrumbs` component with a series of `commandNavigationItem` components as children.

To create a breadcrumb:

1. Create a `breadCrumbs` component by dragging and dropping a **Bread Crumbs** component from the Component Palette to the JSF page.

2. By default, breadcrumb links are displayed in a horizontal line. To change the layout to be vertical, in the Property Inspector, expand the **Common** section and set the `orientation` attribute to `vertical`.
3. For each link in the breadcrumb, create a `commandNavigationItem` component by dragging and dropping a **Navigation Item** from the Component Palette 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 Property Inspector, expand the **Common** section and provide a static string outcome of an action or use an EL expression to reference an action method through the `action` property. If you use a string, it must match the navigation metadata set up in the navigation rule for the page created in Step 1. 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 18–11, drag and drop four `navigationPane` components, as shown in Example 18–18.

Example 18–18 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>
```

Note: Similarly, instead of using individual `commandNavigationItem` components, you can bind the `value` attribute of the `breadCrumbs` component to an `XMLMenuModel` implementation, and use one `commandNavigationItem` component in the `nodeStamp` facet of the `breadCrumbs` component to stamp out the items for a page. For information about the `XMLMenuModel` class, see Section 18.6, "Using a Menu Model to Create a Page Hierarchy".

18.7.3 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) is displayed 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 or how to handle child components). For more information about events, see Chapter 5, "Handling Events." For information about using popup dialogs and windows, see Chapter 13, "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 will handle the actual tab removal, as shown in Example 18–19.

Example 18–19 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 ...
    }
  }
}
```

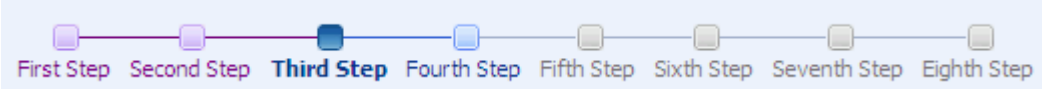
18.8 Using Train Components to Create Navigation Items for a Multi-Step Process

> **Note:** If your application uses the Fusion technology stack or the ADF Controller, then you should use ADF task flows to create the navigation system for your application page hierarchy. For details, see the "Creating a Train" section of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 18–12 shows an example of what a rendered `train` component might look like on a page. Not only does a train 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 18–12 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 be displayed. 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.

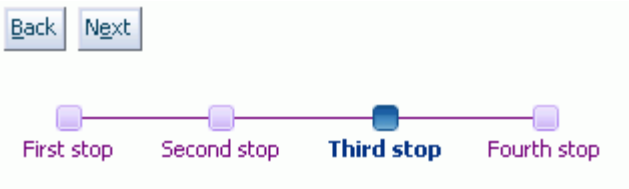
As shown in Figure 18–12, the `train` component provides at least four styles for train stops. The current stop where the user is visiting is indicated by a bold font style in the train stop's label, and a different image for the stop; visited stops before the current stop are indicated by a different 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 a command component (for example, a `commandButton` 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 18–13.

Figure 18–13 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 18–14. 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 18–14 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 "Creating a Train" section of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 10, "Using Tables and Trees"). 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 exposes 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, which you can create by extending the `org.apache.myfaces.trinidad.model.ProcessMenuModel` class. The train behavior controls what stops along the train users can visit while visiting at a 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.

Binding a train component to a train menu model is similar to binding a `navigationPane` component to an `XMLMenuModel` class (described in Section 18.6.3, "How to Bind to the XMLMenuModel in the JSF Page"). However, as long as your `TrainStopModel` implementation returns a `rowData` object, you do not need to

provide the `commandNavigationItem` components for each stop. 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 18–20 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 18.8.3, "How to Bind to the Train Model in JSF Pages".

Example 18–20 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. At page 4 the user jumps back to page 2. Where the user can go next depends on which train behavior is used in the train model.

In Max Visited, from the current page 2 the user can go back to page 1, go ahead to page 3, or jump ahead to page 4. That is, Max Visited allows the user to return to a previous page or advance to any page up to the farthest page already visited. The user cannot jump ahead to page 5 from page 2 because page 5 has not yet been visited.

Given the same situation, in the Plus One behavior the user can only go ahead to page 3 or go back to page 1. That is, Plus One allows the user to return to a previous page or advance one more stop further than the current stop. The user cannot jump ahead to page 4 even though page 4 has already been visited.

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 18.7.1, "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 7, "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 18.8.1, "How to Create the Train Model".
- Configure managed beans for the train model. See Section 18.8.2, "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 18.8.3, "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.

18.8.1 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 (either 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` for implementing a train behavior for the train model

For examples of train model classes, see the `oracle.adfdemo.view.nav.rich` package of the ADF Faces Demonstration application.

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 18–20.

   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 non-interactive. 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 17, "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. `visited` 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 4.2, "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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 18.8.2, "How to Configure Managed Beans for the Train Model"). If the `maxPathKey` value is `null`, then ADF Faces uses the Plus One behavior.

18.8.2 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 create the train model. 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.

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 18.8.1, "How to Create the Train Model". Example 18–21 shows sample managed bean code for train stops in the `faces-config.xml` file.

Example 18–21 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.jspx</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.jspx</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 18.8.1, "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 18–25).

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 18–22.

Example 18–22 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.jspx</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 18–23 shows sample managed bean code for creating the train stop list.

Example 18–23 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 18–22).

Example 18–24 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 18–25 shows sample managed bean code for a train model.

Example 18–25 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.

18.8.3 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.

To bind the train component to the train model:

1. Create a `train` component by dragging and dropping a **Train** from the Component Palette to 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 the following code:

```
<af:train value="#{trainMenuModel}"/>
  <af:trainButtonBar value="#{trainMenuModel}"/>
```

The `trainMenuModel` in the EL expression is the managed bean name for the train model (see Example 18–25).

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 18–26.

Example 18–26 Metadata to Bind a Train Component to the Train Model Bean

```
<af:train value="#{aTrainMenuModel}" var="stop">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem
      text="#{stop.label}"
      action="#{stop.outcome}"
      ...
    </af:commandNavigationItem>
  </f:facet>
</af:train>
```

19

Creating and Reusing Fragments, Page Templates, and Components

This chapter describes how you can create reusable content and then use that content to build portions of your JSF pages or entire pages.

This chapter includes the following sections:

19.1 Introduction to 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 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 19.2, "Using Page Fragments."

- 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. For details about creating and using page templates, see Section 19.3, "Using Page Templates," and Section 19.3.3, "How to Create JSF Pages Based on Page Templates."
- 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. For details about creating and using declarative components, see Section 19.4, "Using Declarative Components."

  Tip: If your application uses the 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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

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 3.5, "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 19.5, "Adding Resources to Pages."

19.2 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 4.6, "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 folder hierarchy of the application.
- `contentViews.jspx`: Contains the content for the folder selected in the navigator pane.

Example 19–1 shows the abbreviated code for the included `header.jspx` page fragment. Note that it does not contain an `f:view` or `document` tag.

Example 19–1 header.jspx Page Fragment

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          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>
</jsp:root>
```

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 19–2, which is abbreviated code from the `index.jspx` page.

Example 19–2 File Explorer Index JSF Page Includes Fragments

```
<?xml version='1.0' encoding='utf-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          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>
</jsp:root>
```

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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Like complete JSF pages, page fragments can also be based on a page template, as shown in Example 19–3. For information about creating and applying page templates, see Section 19.3, "Using Page Templates," and Section 19.3.3, "How to Create JSF Pages Based on Page Templates."

Example 19–3 Page Fragment Based on a Template

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
          xmlns:f="http://java.sun.com/jsf/core">
  <af:pageTemplate viewId="/someTemplateDefinition.jspx">
  .
  .
  .
  </af:pageTemplate>
</jsp:root>
```

19.2.1 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 Application Navigator in JDeveloper.

To create a page fragment:

1. In the Application Navigator, right-click the folder where you wish to create and store page fragments and choose **New**.
2. In the **Categories** tree, select the **JSF** node, in the **Items** pane select **JSF Page Fragment**, and click **OK**.
3. Enter a name for the page fragment file.
4. 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`.
5. You can have your fragment pre-designed for you by using either a template or a Quick Start Layout.
   - If you want to create a page fragment based on a page template, select the **Page Template** radio button and then select a template name from the

dropdown list. For more information about using page templates, see Section 19.3.3, "How to Create JSF Pages Based on Page Templates."

- If you want to use a Quick Start Layout, select the **Quick Start Layout** radio button and then click 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 8.2.3, "Using Quick Start Layouts."

When the page fragment creation is complete, JDeveloper displays the page fragment file in the visual editor.

6. To define the page fragment contents, drag and drop the desired components from the Component Palette onto the page.

   You can use any ADF Faces or standard JSF component, for example `table`, `panelHeader`, or `f:facet`.

Example 19–4 shows an example of a page fragment that contains a menu component.

Example 19–4 Page Fragment Sample

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <!-- 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>
</jsp:root>
```

19.2.2 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 19–5 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 19–5 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.

19.2.3 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 Component Palette or the Application Navigator.

19.2.3.1 Adding a Page Fragment Using the Component Palette

You can use the `jsp:include` tag to include the desired page fragment file

To add a page fragment using the Component Palette:

1. In the Component Palette, use the dropdown menu to choose **JSP**.
2. Add a `jsp:include` tag by dragging and dropping **Include** from the Component Palette.
3. 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 more information, click **Help** in the dialog.

19.2.3.2 Adding a Page Fragment Using the Application Navigator

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

To add a page fragment using the Application Navigator:

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

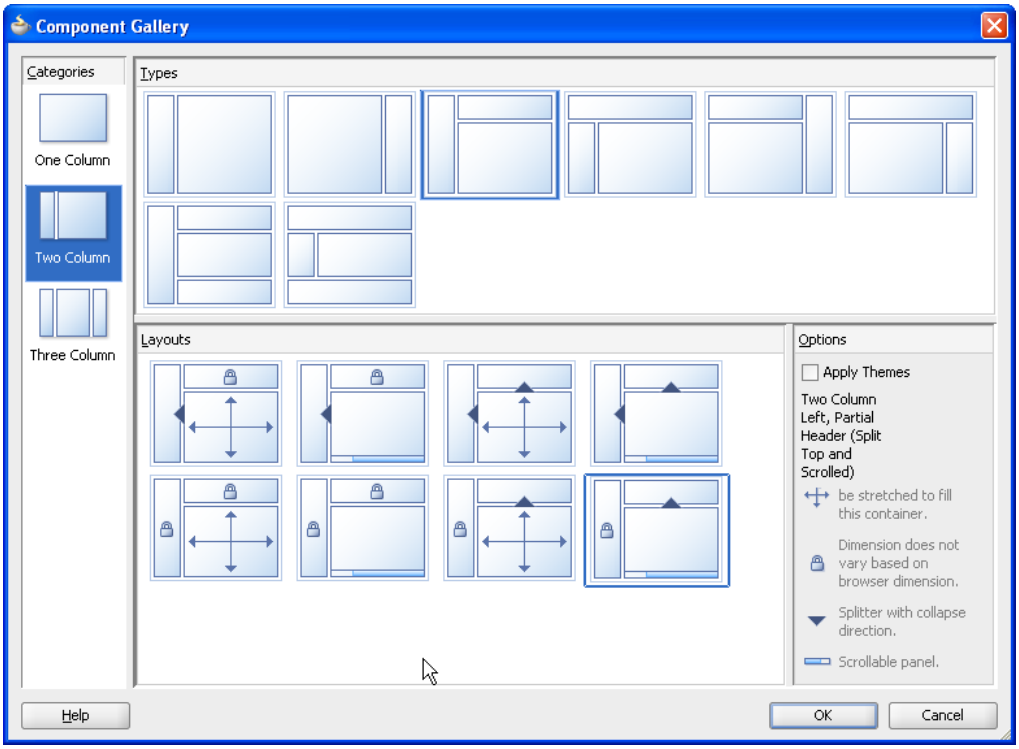
19.2.4 What Happens at Runtime: Resolving Page Fragments

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.

19.3 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 19–1 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 8, "Organizing Content on Web Pages."

Figure 19–1 Quick Layouts



To use page templates in an application, you first create a page template definition. Page template definitions must be JSF documents written in XML syntax (with the file extension of `.jspx`) 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. Either the template or the page that uses the template must contain the `document` tag, but they cannot both contain the 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 19–6.

Example 19–6 Facet Definition in a Template

```
<facet>
  <description>
    <![CDATA[Area to put a commandLink 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 19–7, 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 19–7 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:** To avoid component ID collisions at runtime, 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 19–7 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 19–8

Example 19–8 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.

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 19–9.

Example 19–9 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 21.3, "Manually Defining Resource Bundles and Locales."

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 19.5, "Adding Resources to Pages."

19.3.1 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.

To create a page template definition:

1. In the Application Navigator, right-click the folder where you wish to create and store page templates and choose **New**.
2. In the **Categories** tree, select the **JSF** node, in the **Items** pane select **JSF Page Template**, and click **OK**.
3. 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.

4. Accept the directory name for the template definition, or choose a new location.
5. Enter a Page Template name for the page template definition.
6. If you want to use one of the predefined quick layouts, select **Use a Quick Start Layout** and click **Browse** to select the one you want to use.
7. To add named facets, click the **Facet Definitions** tab and click the **Add** icon.

   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.
8. To add attributes, click the **Attributes** tab and click the **Add** icon.

   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 12) 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.
9. If the page template contents use ADF Model data bindings, select the **Create Associated ADFm Page Definition** checkbox, and click **Model Parameters** to add one or more model parameters. For information about using model parameters and ADF Model data bindings, see the "Using Page Templates" section of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

   Once you complete the wizard, JDeveloper displays the page template definition file in the visual editor. Example 19–10 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 Property Inspector.

> **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 19–10 Component Metadata in Page Template Definition

```
<af:pageTemplateDef var="attrs">
  <af:xmlContent>
    <component xmlns="http://xmlns.oracle.com/adf/faces/rich/component">
      <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>
```

10. Drag a component from the Component Palette and drop it onto the page in the visual editor.

    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 can add any number of components to the layout section. 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 8, "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 19.4, "Using Declarative Components." For information about using databound components in page templates, see the "Using Page Templates" section of the

Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework.

11. Within those components (in the layout section) where content can eventually be inserted by page authors using the template, drag **FacetRef** from the Component Palette and drop it in the desired location on 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 FacetRef 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.

12. 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 19–10, 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}">
    ```

13. 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="secondaryDecoration">
      <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 19.2.3, "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.

19.3.2 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

The first time you use the wizard to create a JSF 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 19–11 shows an example of the `pagetemplate-metadata.xml` file.

Example 19–11 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 Application Navigator, 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.

19.3.3 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. For information about deploying a project, see the "Reusing Application Components" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for 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.

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

1. Follow the instructions in Section 2.4.1, "How to Create JSF JSP Pages" to open the Create JSF Page dialog. In the dialog, select a page template to use from the **Use Page Template** dropdown list.

   Tip: Only page templates that have been created using the template wizard in JDeveloper are available for selection. If the **Use Page Template** dropdown list is disabled, this means no page templates are available in the project where you are creating new pages.

   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 Property Inspector, 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 Component Palette 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 *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

4. In the Structure window, select **af:pageTemplate**. Then, in the Property Inspector, 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 19–12).

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 Property Inspector 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 19.5, "Adding Resources to Pages."

19.3.4 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 19–12. 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 19–12 JSF Page that References a Page Template

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <jsp:directive.page contentType="text/html;charset=windows-1252"/>
  <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>
```

19.3.5 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

19.3.6 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 with client-side events in template-based pages, you must include the template's ID when using code to locate a component. For more details, see Section 5.3.7, "What You May Need to Know About Using Naming Containers."

19.4 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:** 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. For more information about scopes, see Section 4.6, "Object Scope Lifecycles."

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 `panelHeader` 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 `panelHeader` 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 19.4.1, "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.

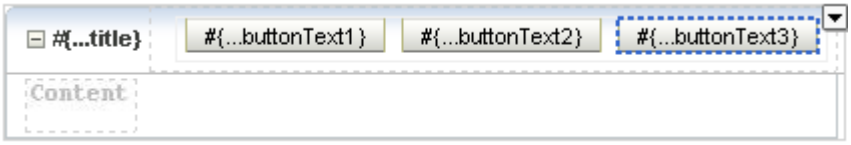
To use declarative components in a project, you first must deploy the library that contains the declarative component as an ADF Library. You can then add the deployed ADF Library JAR to the project's properties, which automatically inserts the JSP tag library or libraries into the project's properties. Doing so allows the component(s) to be displayed in the Component Palette 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 `toolbarButton` 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 19–2 shows how such a declarative component would look in the visual editor.

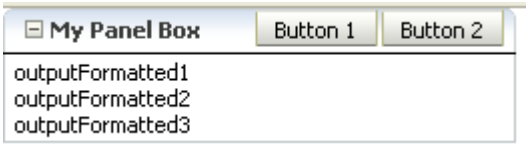
Figure 19–2 Declarative Component in the Visual Editor



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 19–3.

Figure 19–3 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 19.5, "Adding Resources to Pages."

19.4.1 How to Create a Declarative Component

JDeveloper simplifies creating declarative component definitions by providing the Create JSF 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.

To create a declarative component definition:

1. In the Application Navigator, right-click the folder where you wish to create and store declarative components and choose **New**.
2. In the **Categories** tree, select the **JSF** node, in the **Items** pane select **JSF Declarative Component**, and click **OK**.
3. 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 Component Palette, 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 `SampleName1`.

   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 `/ViewController/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 ellipses button 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 Property Inspector.

11. Drag a component from the Component Palette and drop it 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 in the *Oracle Fusion Middleware Fusion Developer's Guide for 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.

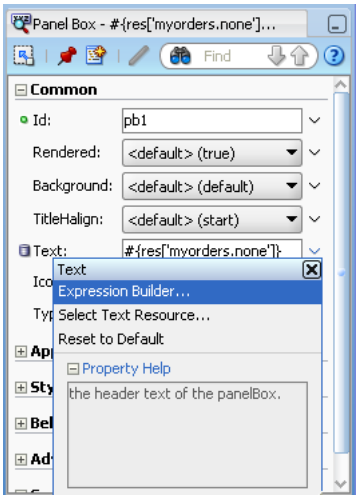
    When you drag **FacetRef** from the Component Palette and drop it in the desired location on the page, JDeveloper displays the Insert FacetRef dialog. In that dialog, select a facet name from the dropdown list, or enter a facet name. If you enter a facet name that is not already defined in the component metadata of the 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 Property Inspector and the Expression Builder to bind component attribute values to the created attributes.

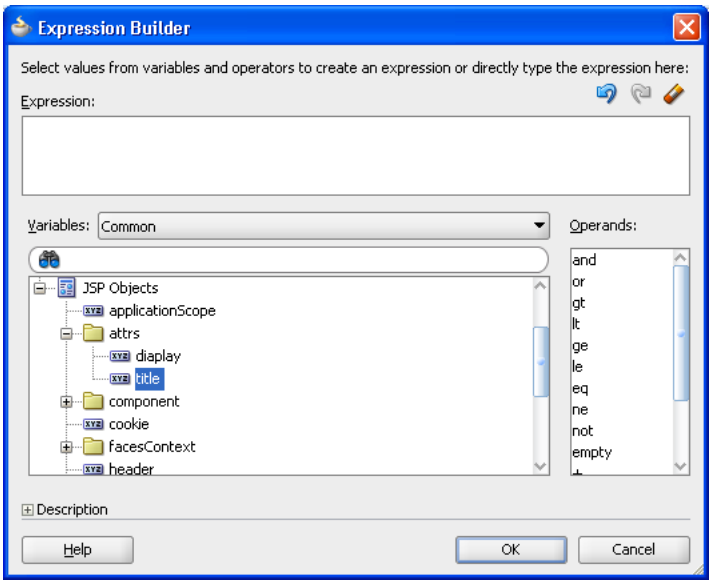
    For example, if you have defined a `title` attribute and added a `panelBox` as a component, you might use the dropdown menu next to the text attribute in the Property Inspector to open the Expression Builder, as shown in Figure 19–4.

Figure 19–4 Opening the Expression Builder for an Attribute in the Property Inspector



    In the Expression Builder, you can expand the **JSP Objects** > **attrs** node to select the created attribute that should be used for the value of the attribute in the Property Inspector. For example, Figure 19–5 shows the **title** attribute selected in the Expression Builder. Click the **Insert Into Expression** button and then click **OK** to add the expression as the value for the attribute.

Figure 19–5 Expression Builder Displays Created Attributes



14. To specify the methods that command 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**.

19.4.2 What Happens When You Create a Declarative Component

When you first use the Create JSF 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. By default the value of `componentVar` is `component`.

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 19–13 shows abbreviated code for the declarative component shown in Figure 19–2.

Example 19–13 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>component</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 19–14 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 19–14 Components in a Declarative Component

```
<af:panelBox text="#{attrs.title}" inlineStyle="width:25%;">
  <f:facet name="toolbar">
    <af:group>
      <af:toolbar>
        <af:commandToolbarButton text="#{attrs.buttonText1}"
                                 actionListener="#{component.handleMethod1}"
                                 rendered="#{attrs.display1}"/>
        <af:commandToolbarButton text="#{attrs.buttonText2}"
                                 rendered="#{attrs.display2}"
                                 actionListener="#{component.handleMethod2}"/>
        <af:commandToolbarButton 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 19–15 shows an example of the `declarativecomp-metadata.xml` file.

Example 19–15 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>
      /dcomponentLib1
    </taglib-uri>
    <taglib-prefix>
      dc
    </taglib-prefix>
  </declarativecomp-taglib>
</declarativeCompDefs>
```

> **Note:** When you rename or delete a declarative component in the Application Navigator, 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.

19.4.3 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. For instructions on how to deploy a project to an ADF Library JAR, see the "Reusing Application Components" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for 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:

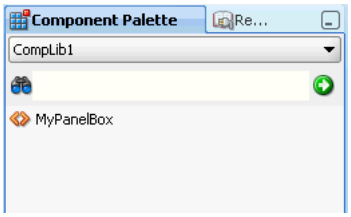
- A component tag class (for example, the `componentDef1Tag.class`) for each declarative component definition (that is, for each `componentDef` component)
- One or more JSP TLD files for the declarative components, using information from the project's `declarativecomp-metadata.xml` file

To use declarative components in a consuming project, you add the deployed ADF Library JAR to the project's properties. For instructions on how to add an ADF Library JAR, see the "Reusing Application Components" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*. By adding the deployed JAR, 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 Component Palette.

19.4.4 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 Component Palette, 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 19–6 shows the page in the Component Palette for a library with a declarative component.

Figure 19–6 Component Palette 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 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, see the "Reusing Application Components" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

To use declarative components in a JSF page:

1. In the Application Navigator, double-click the JSF page (or JSF page template) to open it in the visual editor.
2. In the Component Palette, select the declarative components tag library name from the dropdown list. Drag and drop the desired declarative component onto 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.

> **Note:** If you want to use ADF Model layer bindings as values for the attributes, then to create these bindings manually by using the Expression Builder to locate the needed binding property.

3. Add components by dragging and dropping components from the Component Palette 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 Property Inspector 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 19.5, "Adding Resources to Pages."

19.4.5 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 19–16 shows the code for the MyPanelBox declarative component to which a user has added a panelGroupLayout component that contains three outputFormatted components.

Example 19–16 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>
```

19.4.6 What Happens at Runtime

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.

19.5 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.

19.5.1 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.

To add resources:

1. From the **Operations** section of the Component Palette, drag and drop a **Resource** tag anywhere onto the consuming page.
2. In the Insert Resource dialog, select either `css` or `javascript`.
3. In the Property Inspector, enter the URI of the resource as the value for the `source` attribute. Start the URI with a single forward slash (`/`) if the URI should be context relative. Start the URI with two forward slashes if the URI should be server relative. If you start the URI with something other than one or two slashes, the URI will be resolved relative to URI location in the browser

19.5.2 What Happens at Runtime: Adding 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.

20

Customizing the Appearance Using Styles and Skins

This chapter describes how to change the appearance of your application by changing style properties using ADF Faces skins and component style attributes.

This chapter includes the following sections:

20.1 Introduction to Skins, Style Selectors, and Style Properties

JDeveloper supports two options for applying style information to your ADF Faces components:

- Build a *skin* and a cascading style sheet (CSS) using defined *style selectors* and configure your ADF application to use the skin and style sheet.
- Use *style properties* to override the style information from the skin CSS to set specific instances of component display.

ADF Faces components delegate the functionality of the component to a component class, and 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. You cannot customize ADF Faces renderers. However, you can customize how components display using skins.

If you do not wish to change ADF Faces components throughout the entire application, you can choose to change the styles for the instance of a component on a page. You can also programmatically set styles conditionally. For example, you may want to display text in red only under certain conditions. For more information, see Section 20.4, "Changing the Style Properties of a Component".

The File Explorer application allows you to select several skins from a dropdown list. It provides several CSS files to support skin selection. For more information, see Section 1.4.3, "Overview of the File Explorer 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 W3C web site at:

`http://www.w3.org/`

> **Note:** The 11g Release 2 (11.1.2.0.0) introduced the ADF Skin Editor. Using this standalone product, you can visually create and modify skins for ADF Faces applications. The ADF Skin Editor provides a range of features that simplify the process of creating a skin. For more information, including how to install the ADF Skin Editor, see the Release Downloads for Oracle ADF 11g page at `http://www.oracle.com/technetwork/developer-tools/adf/downloads/index.html`. For information about using the ADF Skin Editor, see the *Oracle Fusion Middleware Skin Editor User's Guide for Oracle Application Development Framework*.

20.1.1 ADF Faces Skins

A *skin* is a style sheet based on the CSS 3.0 syntax specified in one place for an entire application. Instead of providing a style sheet for each component, or inserting a style sheet on each page, you can create one skin for the entire application. Every component automatically uses the styles as described by the skin. You do not have to make design-time changes to JSF pages to change their appearance when you use a skin. The skin allows you to globally change the appearance of ADF Faces components.

Existing ADF Faces applications use the skin that the application was configured to use when the application was created. For example, if you created an application using Oracle ADF 11*g* (11.1.1.2.0), the application uses the `fusion` skin. Applications created with subsequent releases use skins that extend this skin. If you upgrade an application, the application continues to use the skin that it was configured to use when first created. You edit the `trinidad-config.xml` file, as described in Section 20.2.4, "How to Configure an Application to Use a Custom Skin," if you want your application to use another skin.

You can create your own custom skin by extending one of the skins provided by ADF Faces. For more information, see Section 20.2.1, "How to Add a Custom Skin to an Application." Create or edit the `trinidad-skins.xml` file, as described in Section 20.2.3, "How to Register a Custom Skin," in addition to editing the `trinidad-config.xml` file if you want your application to use a custom skin that you created.

ADF Faces provides the following skins for use in your applications:

- `simple`: Contains only minimal formatting.
- `blafplus-medium`: Provides a modest amount of styling. This style extends the `simple` skin.
- `blafplus-rich`: This skin extends the `blafplus-medium` skin. Provides more styling than the `blafplus-medium` skin. For example, graphics in the `blafplus-rich` skin have rounded corners.
- `fusion`: Defines the default styles for ADF Faces components. This skin provides a significant amount of styling.
- `fusion-11.1.1.3.0`: Modifies the `fusion` skin to make the hierarchy structure in certain components that render tabs clearer. These components are

panelTabbed, navigationPane (attribute hint="tabs"), and decorativeBox. This skin also defines a more subtle background image for disclosed panelAccordion component panes to make text that appears in these panes easier to read.

- fusionFx-v1: This skin extends from the fusion-11.1.1.3.0 skin. If you create a custom skin that extends any of the skins provided by ADF Faces, you need to register it in the trinidad-skins.xml file. Use the following values in the trinidad-skins.xml file if you extend the fusionFx-v1 skin:

```
<skin>
   <id>yourSkin.desktop</id>
   <family>yourSkinFamily</family>
   <extends>fusionFx-v1.desktop</extends>
</skin>
```

Use the following value in the trinidad-config.xml file if you want your application to use the fusionFx-v1 skin:

```
<skin-family>fusionFx</skin-family>
```

The fusionFx-v1 contains design improvements and changes to address a number of issues. Specifically, it adds:

– A background color to the .AFMaskingFrame global style selector to prevent the display of content from an underlying frame when an inline popup displays in certain browsers.

– A boolean ADF skin property, -tr-stretch-dropdown-table, for the inputComboboxListOfValues component. This property determines whether the table in the dropdown list stretches to show the content of the table columns or limits the width of the table to the width of the input field in the inputComboboxListOfValues component.

– The inlineFrame component displays an image that serves as a loading indicator until the browser determines that the frame's contents have been loaded.

  You can implement this functionality in a custom skin that you create. The af|inlineFrame selector has "busy" and "flow" pseudo-classes that enable you to do this. The inlineFrame component only generates an IFrame element when the parent component does not stretch the inlineFrame component (the inlineFrame component is flowing). Use af|inlineFrame:busy:flow to define a background-image style that references a loading indicator. When the parent component stretches the inlineFrame component, the generated content is more complex. This complexity allows you define a content image URL using the af|inlineFrame::status-icon and an optional additional background-image using the af|inlineFrame::status-icon-style. It also allows you to reuse images that other component selectors use. For example, the carousel component's af|carousel::status-icon and af|carousel::status-icon-style selectors. Use skinning aliases to reuse these images.

  The following global selectors have also been introduced that you can use if you implement this functionality in your ADF skin:

  * .AFBackgroundImageStatus:alias: use to reference the background image used in af|inlineFrame::busy:flow.

    * `.AFStatusIcon:alias` use to reference the `af|carousel::status-icon` and `af|inlineFrame::status-icon`.
    * `.AFStatusIconStyle:alias` use to reference the `af|carousel::status-icon-style` and `af|inlineFrame::status-icon-style`.

    A resource key (`af_inlineFrame.LABEL_FETCHING`) defines the string to display for the `inlineFrame` component's loading icon.

- `fusionFx-v1.1`: This skin extends from the `fusionFx-v1` skin. It adds supports for the ability to clear Query-By-Example (QBE) filters in an `af:table` component.

  If you create a custom skin that extends any of the skins provided by ADF Faces, you need to register it in the `trinidad-skins.xml` file. Use the following values in the `trinidad-skins.xml` file if you want to extend the `fusionFx-v1.1` skin:

  ```
  <skin>
     <id>yourSkin.desktop</id>
     <family>yourSkinFamily</family>
     <extends>fusionFx-v1.1.desktop</extends>
     ...
  </skin>
  ```

  Use the following value in the `trinidad-config.xml` file if you want your application to use the `fusionFx-v1.1` skin:

  ```
  <skin-family>fusionFx</skin-family>
    <skin-version>v1.1<skin-version>
  ```

- `fusionFx-v1.2`: This skin extends from the `fusionFx-v1.1` skin. It contains a number of user interface enhancements including optimizations for when your application renders in a touch screen device.

  Use the following values in the `trinidad-skins.xml` file if you want to extend the `fusionFx-v1.2` skin.

  ```
  <skin>
     <id>yourSkin.desktop</id>
     <family>yourSkinFamily</family>
     <extends>fusionFx-v1.2.desktop</extends>
     ...
  </skin>
  ```

  Use the following value in the `trinidad-config.xml` file if you want your application to use the `fusionFx-v1.2` skin:

  ```
  <skin-family>fusionFx</skin-family>
    <skin-version>v1.2<skin-version>
  ```

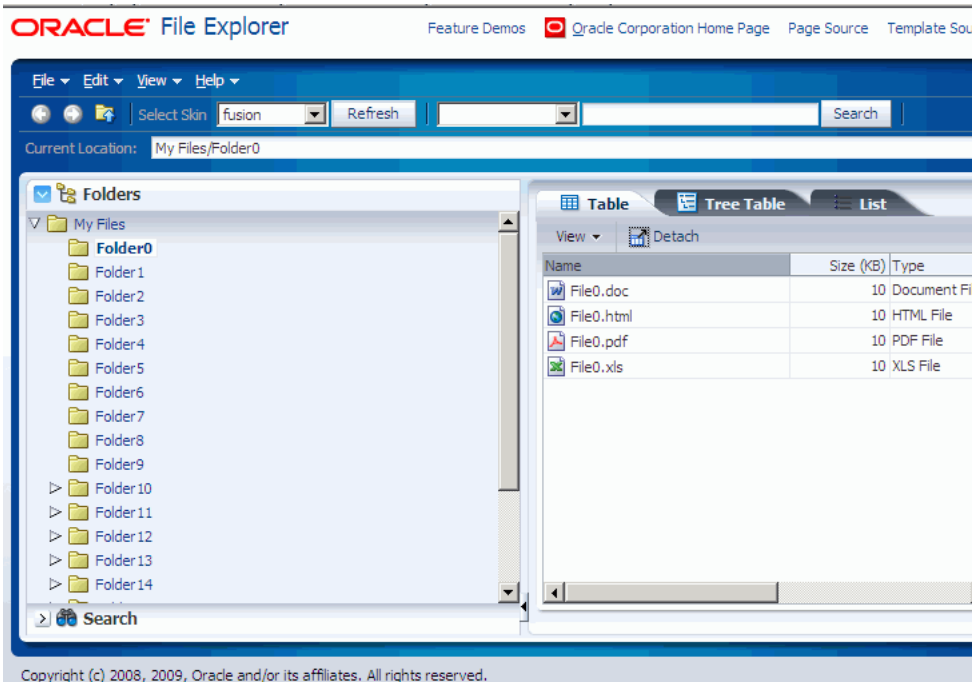
- Projector skins: ADF Faces provides skins that define styles for an application that you want to demonstrate to an audience using a projector. Each projector skin modifies a number of elements in its parent skin so that an application renders appropriately when displayed using table-top projectors (particularly older models of projector). For example, the `fusion-projector` skin modifies a number of elements in the `fusion` skin. These skins are useful if the audience is present at the same location as the projector. They may not be appropriate for an audience that views an application online through a web conference. ADF Faces

provides the projector skins as a download from the Oracle Technology Network (OTN) web site.

Figure 20–1 shows the default `fusion` skin applied to the File Explorer Application index page.

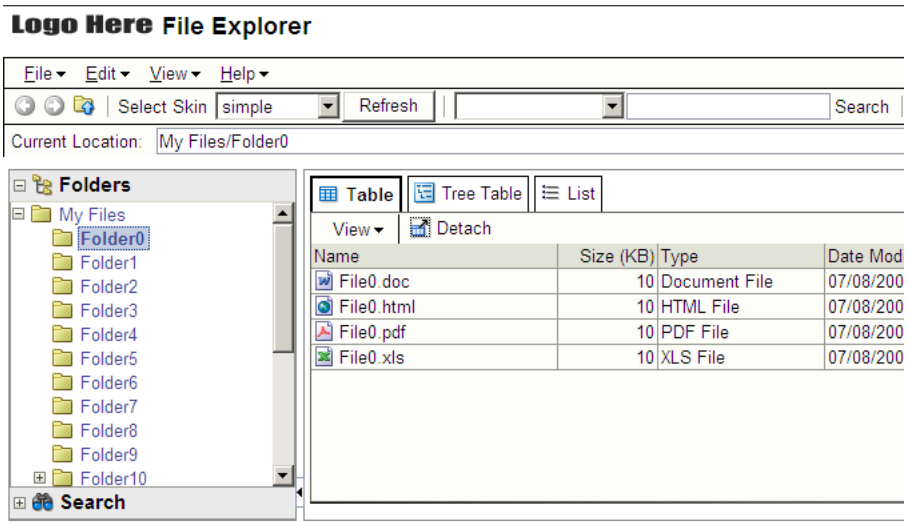
Note: The syntax in a skin style sheet is based on the CSS 3.0 specification. However, many browsers do not yet adhere to this version. At runtime, ADF Faces converts the CSS to the CSS 2.0 specification.

Figure 20–1 Index Page Using the Fusion Skin



ADF Faces also provides the `simple` skin, shown in Figure 20–2 as applied to the File Explorer Application index page.

Figure 20–2 Index Page Using the Simple Skin



Skins provide more options than setting standard CSS styles and layouts. The skin's CSS file is processed by the skin framework to extract skin properties and icons and register them with the `Skin` object. For example, you can customize the skin file using rules and pseudo classes that are supported by the skinning framework. Supported rules and pseudo classes include `@platform`, `@agent`, `@accessibility-profile`, `:rtl`, and `@locale`. For more information, see Section 20.1.2, "Skin Style Selectors".

20.1.2 Skin Style Selectors

Style sheet rules include a style selector, which identifies an element, and a set of style properties, which describe the appearance of the components. ADF Faces components include two categories of skin style selectors:

- Global selectors

  Global selectors determine the style properties for multiple ADF Faces components. If the global selector name ends in the `:alias` pseudo-class, then the selector is most likely included in other component-specific selectors and will affect the skin for more than one component. For example, most, if not all, components use the `.AFDefaultFontFamily:alias` definition to specify the font family. If your skin overrides this selector with a different font family, that change will affect all the components that have included it in their selector definition. Example 20–1 shows the global selector for the default font family for ADF Faces components in an application.

Example 20–1 Global Selector for Default Font Family

```
.AFDefaultFontFamily:alias {
       font-family: Tahoma, Verdana, Helvetica, sans-serif;
}
```

- Component selectors

  Component-specific selectors are selectors that can apply a skin to a particular ADF Faces component. Example 20–2 shows the selector set to red as the background color for the content area of the `af:inputText` component.

Example 20–2 af:inputText Component Selector

```
af|inputText::content {
         background-color: red;
}
```

Each category may include one or more of these *types* of ADF Faces skin selectors:

- Standard selectors

  Standard selectors are those that directly represent an element that can have styles applied to it. For example, `af|body` represents the `af:body` component. You can set CSS styles, properties, and icons for this type of element.

- Selectors with pseudo-elements

  Pseudo-elements are used to denote a specific area of a component that can have styles applied. Pseudo-elements are denoted by a double colon followed by the portion of the component the selector represents. For example, `af|chooseDate::days-row` provides the styles and properties for the appearance of the dates within the calendar grid.

- Icon selectors

  Some components render icons (`<img>` tags) using a set of base icons. These icons can have skins applied even though no entries appear in the CSS source file for the icons in the way, for example, that entries appear for the `background-image` CSS property. Instead, the icons are registered with the `Skin` object for use by the renderer. As no entries for an icon selector appear in the CSS source file that a browser interprets, you cannot create containment selector definitions for an icon definition. You can only create a containment selector definition for items that have an entry in the CSS source file.

  Icon selectors are denoted by `-icon` for component selectors and `Icon:alias` for global selectors. For example, the `af:inputDate` component has a changed icon that can have a skin using the selector `af|inputDate::changed-icon`. The changed icon can also be globally set for all components using that icon with the global selector `.AFChangedIcon:alias`. For more information, see Section 20.3.2, "How to Apply Skins to Icons".

- Resource strings

  The text rendered by ADF Faces components is translatable. The text is abstracted as a resource string that has skins applied. For example, `af_dialog.LABEL_OK` is a resource string for the text label of an `af:dialog` component when the **OK** button has been configured. Resource strings do not have skins in the CSS skin file, but in a resource bundle referenced from the skin definition file in the `trinidad-skins.xml` file using the `<bundle-name>` parameter. You can also use the `<translation-source>` parameter for an EL binding to point to a `Map` or `ResourceBundle`. For more information, see Section 20.3.1, "How to Apply Skins to Text".

- Selectors with style properties

  Skin style properties allow you to customize the rendering of a component throughout the application. A CSS property is stored with a value in the `Skin` object and is available when the component is being rendered. For example, in `af|breadCrumbs{-tr-show-last-item: false}`, the skin property `-tr-show-last-item` is set to hide the last item in the `af:breadCrumbs` navigation path.

The CSS specification defines pseudo-classes such as `:hover` and `:active` that can apply to almost every component. ADF Faces provides additional pseudo-classes for specialized functions. Pseudo-classes are denoted in the selector by a colon followed by the class definition. The following are common pseudo-classes used by ADF Faces style selectors:

- Alias: The `:alias` pseudo-class is a special type of class that serves as a syntax aid to organize code in your skin file. You can, for example, use it to set styles for more than one component or more than one portion of a component. You can also create your own alias classes that you can then include on other selectors. For example, you can define an alias pseudo-class (`.AFLabel:alias`) where you define label colors for a number of form components. Subsequent changes to the alias pseudo-class impact all components referenced by the alias pseudo-class.

  ```
  af|inputText::label,
  af|inputChoice::label,
  af|selectOneChoice::label {-tr-rule-ref: ".AFLabel:alias"}
  .AFLabel:alias { color: blue }
  ```

  The `.AFLabel:alias` pseudo-class has color set to blue, but you can change all the component's label color to red by simply changing `.AFLabel:alias`:

  ```
  .AFLabel:alias {color: red}
  ```

  For more information, see Section 20.3.5, "How to Create a Custom Alias".

- Drag and drop: The two pseudo-classes available are `:drag-source` applied to the component initiating the drag and removed once the drag is over, and `:drop-target` applied to a component willing to accept the drop of the current drag.

- Standard: In CSS, pseudo-classes like `:hover`, `:active`, and `:focus` are considered states of the component. This same concept is used in applying skins to components. Components can have states like `read-only` or `disabled`. When states are combined in the same selector, the selector applies only when all states are satisfied.

- Right-to-left: Use this pseudo-class to set a style or icon definition when the browser is in a right-to-left language. Another typical use case is asymmetrical images. You will want the image to be flipped when setting skin selectors that use the image in a right-to-left reading direction. Be sure to append the `:rtl` pseudo-class to the very end of the selector and point it to a flipped image file. For example, the end image of the `panelBox` component will be the `panelBoxStart.png` file when the browser is set to right-to-left. The `panelBox` end image in right-to-left is the same as the flipped left-to-right `panelBox` start image.

  ```
  af|panelBox::medium af|panelBox::top-end:rtl {
            background-image: url(/skins/purple/images/panelBoxStart.png);
            width:8px;
            height:8px
  }
  ```

  You can also use `:rtl` to apply to skin icons. For more information, see Section 20.3.2, "How to Apply Skins to Icons".

- Inline editing: This pseudo-class is applied when the application activates a component subtree for editing in the browser. For example, `:inline-selected` is a pseudo-class applied to currently selected components in the active inline-editable subtree.

- Message: This pseudo-class is used to set component-level message styles using CSS pseudo-classes of `:fatal`, `:error`, `:warning`, `:confirmation`, and `:info`. For more information, see Section 20.3.3, "How to Apply Skins to Messages".

You may not want your selector's CSS properties to be applied to all browsers, all platforms, all locales, and both reading-directions. For example, you may need to add some padding in Internet Explorer that you do not need on any other browser. You may want the font style to be different on Windows than it is on other platforms. To style a selector for a particular user environment, put that skinning information inside a skinning framework rule or `:rtl` pseudo-class. The skinning framework picks the styles based on the `HTTP` request information, such as agent and platform, and merges them with the styles without rules. Those CSS properties that match the rules get merged with those outside of any rules. The most specific rules that match a user's environment take precedence. The skinning framework currently supports these rules and pseudo-classes:

- `@platform` and `@agent`

  Define platform styles using `@platform` and browser styles using `@agent`.

  The supported values to set a platform-specific style are `windows`, `macos`, `linux`, `solaris`, and `ppc`. For a browser agent-specific style, the supported values are `ie`, `mozilla`, `gecko`, `webkit` (maps to safari), `ice`, and `email`.

  In this example, the content area of the `af:inputText` component is set to the color pink for versions 7 and 8 of Internet Explorer, and set to version 1.9 of `gecko` on Windows and Linux platforms:

```
@platform window, linux {
  @agent ie and (version: 7) and (version: 8), gecko and (version: 1.9) {
   af|inputText::content {background-color:pink
   }
  }
}
```

  Note that the following syntax examples results in the same behavior:

```
@agent ie and (version: 7.*)
@agent ie and (version: 7)
```

  In order to specify only version 7.0.x of Internet Explorer, use the following syntax:

```
@agent ie and (version: 7.0)
```

  There is currently no syntax to specify a range of versions.

  You can also use the `@agent` rule to determine styles to apply to agents that are touchscreen devices. The following examples show the syntax that you write in a custom skin file to configure this capability.

```
@agent (touchScreen:none) {
   /* Styles that should not render on touchscreen devices. */
}

@agent (touchScreen:single) {
   /* Styles specific for a touchscreen device with single touch. */
}

@agent (touchScreen:multiple) {
   /* Styles specific for a touchscreen with multiple touch. */
}
```

```
@agent (touchScreen) {
   /* Touchscreen specific styles for all touchscreen devices: both single and
multiple touch. */
}
```

For more information about creating applications to render in touchscreen devices, see Appendix D, "Creating Web Applications for Touch Devices Using ADF Faces."

- @accessibility-profile

  Define `@accessibility-profile`, which defines styles for *high-contrast* and *large-fonts* accessibility profile settings from the `trinidad-config.xml` file.

  The *high-contrast* value would be for cases where background and foreground colors need to be highly contrasted with each other. The *large-fonts* value would be for cases where the user must be allowed to increase or decrease the text scaling setting in the web browser. Defining *large-fonts* does not mean that the fonts are large, but rather that they are scalable fonts or dimensions instead of fixed pixel sizes.

  ```
  <!-- Enable both high-contrast and large-fonts content -->
    <accessibility-profile>high-contrast large-fonts</accessibility-profile>
  ```

- `:rtl`

  Use the `:rtl` pseudo-class to create a style or icon definition when the browser is displaying a right-to-left language.

- `@locale`

- Suppress *skin* styles with the `-tr-inhibit` skin property.

  Suppress or reset CSS properties inherited from a base skin with the `-tr-inhibit` skin property. For example, the `-tr-inhibit:padding` property will remove any inherited padding. Remove (clear) all inherited properties with the `-tr-inhibit:all` property. The suppressed property name must be matched exactly with the property name in the base skin.

- Merge styles with the `-tr-rule-ref` property.

  Create your own alias and combine it with other style selectors using the `-tr-rule-ref` property. For more information, see Section 20.3.5, "How to Create a Custom Alias".

- Alter themes of child components with the `-tr-children-theme` property.

  For more information, see Section 20.3.4, "How to Apply Themes to Components".

Example 20–3 shows several selectors in the CSS file that will be merged together to provide the final style.

Example 20–3 Merging of Style Selectors

```
/** For IE and Gecko on Windows, Linux and Solaris, make the color pink. **/
@platform windows, linux, solaris
        {
          @agent ie, gecko
          {
            af|inputText::content {background-color:pink}
          }
        }

af|someComponent {color: red; width: 10px; padding: 4px}
```

```
/* For IE, we need to increase the width, so we override the width.
 We still want the color and padding; this gets merged in. We want to add
 height in IE.  */

@agent ie
        {
          af|someComponent {width: 25px; height: 10px}
        }
/* For IE 7 and 8, we also need some margins.*/
@agent ie (version: 7) and (version: 8)
        {
          af|someComponent {margin: 5px;}
        }

/* For Firefox 3 (Gecko 1.9) use a smaller margin.*/
@agent gecko (version: 1.9)\
        {
          af|someComponent {margin: 4px;}
        }

/* The following selectors are for all platforms and all browsers. */
/* rounded corners on the top-start and top-end */
/* shows how to use :rtl mode pseudo-class. The start image in ltr mode is the */
 /* same as the end image in the right-to-left mode. */
af|panelBox::medium af|panelBox::top-start,
af|panelBox::medium af|panelBox::top-end:rtl {
          background-image: url(/skins/purple/images/panelBoxStart.png);
          width:8px;
          height:8px
        }

af|panelBox::medium af|panelBox::top-end,
af|panelBox::medium af|panelBox::top-start:rtl {
          background-image: url(/skins/purple/images/panelBoxEnd.png);
          height: 8px;
          width:  8px;
        }
```

The selectors used to apply skins to the ADF Faces components are defined in the "Skin Selectors for Fusion's ADF Faces Components" and "Skin Selectors for Fusion's Data Visualization Tools Components" topics in JDeveloper's online help.

You can also apply themes as a way to implement look and feel at the component level. For information about themes, see Section 20.3.4, "How to Apply Themes to Components".

For information about defining skin style properties, see Section 20.3, "Defining Skin Style Properties".

20.1.3 Component Style Properties

You can adjust the look and feel of any component at design time by changing the style-related properties, `inlineStyle` and `styleClass`, both of which render on the *root* DOM element. Any style-related property you specify at design time overrides the comparable style specified in the application skin or CSS for that particular instance of the component.

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.

For information about applying component style properties, see Section 20.4, "Changing the Style Properties of a 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 30.4.7, "How to Create a Class for a Renderer".

20.2 Applying Custom Skins to Applications

Custom skins can change the colors, fonts, and even the location of portions of ADF Faces components to represent your company's preferred look and feel. You build the skin by defining style selectors in a CSS file. After you create your custom style sheet, register it as a valid skin in the application, and then configure the application to use the skin. If you versioned multiple ADF skins in the same skin family, as described in Section 20.6, "Versioning Custom Skins," use the `<skin-version>` element to identify the specific version that you want the application to use.

By default, ADF Faces components use the `fusion` skin. Custom skins can extend to any of the ADF Faces skins, `fusion`, `blafplus-rich`, `blafplus-medium`, or `simple`. To create a custom skin, you declare selectors in a style sheet that override or inhibit the selectors in the style sheet being extended. Any selectors that you choose not to override will continue to use the style as defined in that skin.

Extending the `simple` skin does not require inhibiting as many properties as you would if you extended the BLAF Plus skins. For example, the BLAF Plus skins use many different *colors* for style properties, including text, background, and borders. The `simple` skin uses the `:alias` pseudo-class, as in `.AFDarkBackground:alias`, instead of specific colors. Changing a color scheme would require overriding far fewer global skin selectors than component skin selectors that specify multiple colors.

The text used in a skin is defined in a resource bundle. As with the selectors for the `blafplus-rich` skin, you can override the text by creating a custom resource bundle and declaring only the text you want to change. After you create your custom resource bundle, register it with the skin.

You can create and apply multiple skins. For example, you might create one skin for the version of an application for the web, and another for when the application runs on a PDA. Or you can change the skin based on the locale set on the current user's browser. Additionally, you can configure a component, for example an `af:selectOneChoice` component, to allow a user to switch between skins.

While you can bundle the custom skin resources and configuration files with the application for deployment, you can also store skin definitions in a Java Archive (JAR) file and then add it to the deployed application. The advantages to using a JAR file are that the custom skin can be developed and deployed separately from the application, improving consistency in the look and feel, and that skin definitions and image files can be partitioned into their own JAR files, reducing the number of files that may have to be deployed to an application.

The steps to apply a custom skin to your application are the following:

1. Add a custom skin to your application. For details, see Section 20.2.1, "How to Add a Custom Skin to an Application".

2. Register the custom skin. For details, see Section 20.2.2, "How to Register the XML Schema Definition File for a Custom Skin" and Section 20.2.3, "How to Register a Custom Skin".
3. Configure the application to use the custom skin. For details, see Section 20.2.4, "How to Configure an Application to Use a Custom Skin".
4. Deploy a custom skin in a JAR file. For details, see Section 20.7, "Deploying a Custom Skin File in a JAR File".

20.2.1 How to Add a Custom Skin to an Application

To add a custom skin to your application, create a CSS file within JDeveloper, which places the CSS in a project's source file for deployment with the application.

To add a custom skin to an application:

1. In JDeveloper, make sure that **CSS Level 3** and **ADF Faces** are selected. From the main toolbar, choose **Tools > Preferences > CSS Editor**. For Support Level, choose **CSS Level 3** from the dropdown menu, and for Supported Components, select **ADF Faces Extension**.
2. In the Application Navigator, right-click the project that contains the code for the user interface and choose **New** from the context menu.
3. In the New Gallery under **Categories**, expand **Web Tier** and select **HTML**.
4. Double-click the **CSS File** option.
5. In the Create Cascading Style Sheet dialog, enter a name and path for the CSS.
6. Click **OK**.

You can now open the CSS in the CSS editor and define styles for your application. For information about setting ADF Faces component style selectors, see Section 20.3, "Defining Skin Style Properties".

You can also create a CSS outside the context of Oracle JDeveloper and package the CSS with the skin resources into a JAR file. For information about this recommended option, see Section 20.7, "Deploying a Custom Skin File in a JAR File".

20.2.2 How to Register the XML Schema Definition File for a Custom Skin

You need to register the `trinidad-skins.xsd` file with JDeveloper if you plan to register a custom skin, as described in Section 20.2.3, "How to Register a Custom Skin". The `trinidad-skins.xsd` file defines the valid elements for a custom skin.

To register an XML schema definition file:

1. In JDeveloper, select **Tools** > **Preferences**.
2. In the Preferences dialog, select **XML Schemas** in the left pane and click **Add**.
3. In the Add Schema dialog, click **Browse** to navigate to the XML schemas included in your version of JDeveloper.

   The directory path to the XML schemas is similar to the following:

   ```
   JDeveloper_Home/jdeveloper/modules/oracle.adf.view_
   11.1.1/trinidad-impl.jar!/org/apache/myfaces/trinidadinternal
   /ui/laf/xml/schemas/skin/trinidad-skins.xsd
   ```

> **Note:** In the Add Schema dialog, make sure the value in the **Extension** input field is `.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.

4. Click **OK**.

20.2.3 How to Register a Custom Skin

Registering a skin involves creating a file named `trinidad-skins.xml` and populating it with values that identify the skin's ID, family, location, and the custom resource bundle if you are using one.

Before you begin:

Register the XML schema definition file that defines valid elements for the `trinidad-skins.xml` file. For more information, see Section 20.2.2, "How to Register the XML Schema Definition File for a Custom Skin".

To register a custom skin:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **XML**.
3. Select **XML Document from XML Schema** and click **OK**.
4. In the Create XML from XML Schema - Step 1 of 2 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**.
5. In the Create XML Schema - Step 2 of 2 dialog:
   - **Target Namespace**: Select `http://myfaces.apache.org/trinidad/skin`.
   - **Root Element**: Select `skins`.
   - Click **Finish**. The new file automatically opens in the XML Editor.
6. In the XML editor, enter values for the following elements:
   - `<id>`

     A skin is required to have a unique ID. You can also use an EL expression to reference the skin ID. For example, if you want to have different skins for different locales, create an EL expression that selects the correct skin based on its ID. The convention is to put a "desktop" or ".pda" or ".portlet" at the end of the ID, such as "fusion.desktop".

   - `<family>`

     You configure an application to use a particular *family* of skins. This allows you to group skins together for an application, based on the render kit used.

     For example, you can define the `blafplus-rich.desktop` skin and the `blafplus-rich.pda` skin to be part of the `richDemo` family and the system automatically chooses the right skin based on the `render-kit-id`.

     `<skin>`

```
        <id>richdemo.desktop</id>
        <family>richDemo</family>
        <extends>blafplus-rich.desktop</extends>
        <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
        <style-sheet-name>skins/richdemo/richdemo.css</style-sheet-name>
</skin>
<skin>
        <id>richdemo.pda</id>
        <family>richDemo</family>
        <extends>blafplus-rich.pda</extends>
        <render-kit-id>pda</render-kit-id>
        <style-sheet-name>skins/richdemo/richdemo.css</style-sheet-name>
</skin>
```

Note: If you create more than one skin in a particular family of skins, you can version the skins that you create. For more information, see Section 20.6, "Versioning Custom Skins."

- `<extends>`

  You extend a custom skin by using this element. The default value for this element is `simple.desktop`. However, you can extend any skin by using this element.

  For example, you can easily change the font of the entire skin by extending the skin and creating a CSS with the font alias. For example, extend the `fusion.desktop` family as follows:

  ```
  <extends>fusion.desktop</extends>
  <style-sheet-name>skins/fod_skin.css</style-sheet-name>
  ```

  In the CSS, set the alias to change the font for the entire skin:

  ```
  .AFDefaultFontFamily:alias {font-family: Tahoma}
  .AFDefaultFont:alias {font-size: 16px}
  ```

- `<render-kit-id>`

  This value determines which render kit to use for the skin. You can enter one of the following:

  – `org.apache.myfaces.trinidad.desktop`: The skin will automatically be used when the application is rendered on a desktop.

  – `org.apache.myfaces.trinidad.pda`: The skin will be used when the application is rendered on a PDA.

- `<style-sheet-name>`

  This is the URL of the custom style sheet. The style sheet name file is retrieved as a URL object using the following methods:

  – For nonstatic URLs, those that could change after the server has started, the URL is created by calling `new java.new.URL(style-sheet-name)` if `style-sheet-name` starts with `http:`, `https:`, `file:`, `ftp:`, or `jar:`. Otherwise, the URL is created by calling `<FacesContext> <ExternalContext> getResource<style-sheet-name>`. It will add a slash (`/`) to delimit the URL parts if it is not already present. For example, the slash is added between `skins/bigfont/bigfont.css`.

  - If still not retrieved, the URL is created using the `<ClassLoader>` `getResource` in a style-sheet-name format similar to `META-INF/purpleSkin/styles/myPurpleSkin.css`. Once the URL is converted to this format, it can be searched for in JAR files that may contain the style sheet.

- `<bundle-name>`

  This is the resource bundle created for the skin. If you did not create a custom bundle, then you do not need to declare this element. For more information, see Section 20.3.1, "How to Apply Skins to Text".

  Note: If you have created localized versions of the resource bundle, then you need to register only the base resource bundle.

- `<translation-source>`

  This is an EL binding that can point to a `Map` or a `ResourceBundle`. You can use this instead of the bundle name if you would like to be more dynamic in your skin translations at runtime. The `<bundle-name>` tag takes precedence.

Example 20–4 shows the entry in the `trinidad-skins.xml` file for the `mySkin` skin.

Example 20–4 Skin Entry in the trinidad-skins.xml File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
    <skin>
        <id>
            mySkin.desktop
        </id>
        <family>
            mySkin
        </family>
        <extends>blafplus-rich.desktop</extends>
        <render-kit-id>
            org.apache.myfaces.trinidad.desktop
        </render-kit-id>
        <style-sheet-name>
            skins/mySkin/mySkin.css
        </style-sheet-name>
        <bundle-name>
            myBundle
        </bundle-name>
        <translation-source></translation-source>
    </skin>
</skins>
```

7. Save the file.

20.2.4 How to Configure an Application to Use a Custom Skin

You set an element in the `trinidad-config.xml` file that determines which skin to use, and if necessary, under what conditions.

Note: If you do not see the skin, check to see whether or not the `af:document` tag has been added to the page. The `af:document` tag initializes the skin framework to create the CSS and link it to the page.

To configure an application to use a skin:

1. Open the `trinidad-config.xml` file.
2. In the `trinidad-config.xml` file, write entries to specify the value of the `<skin-family>` element for the skin you want to use and, optionally, the `<skin-version>` element.

   Example 20–5 shows the configuration to use for the `mySkin` skin family.

Example 20–5 Configuration to Use a Skin Family

```
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>mySkin</skin-family>
   <skin-version>v2</skin-version>
</trinidad-config>
```

3. To conditionally set the value, enter an EL expression that can be evaluated to determine the skin to display.

   For example, if you want to use the German skin when the user's browser is set to the German locale, and to use the English skin otherwise, you would have the following entry in the `trinidad-config.xml` file:

   ```
   <skin-family>#{facesContext.viewRoot.locale.language=='de' ? 'german' :
   'english'}</skin-family>
   ```

4. Save the file.

During development, after you make changes to the custom skin, you can see your CSS changes without restarting the server by setting the `web.xml` file parameter `org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION` to `true`, as shown in Example 20–6. However, you must always restart the server to see icon and skin property changes.

Example 20–6 web.xml Parameter to Check Skin Changes

```
<context-param>
    <param-name>org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION</param-name>
    <param-value>true</param-value>
</context-param>
```

20.3 Defining Skin Style Properties

The ADF Faces skin style selectors support multiple options for applying skins to a component to create a custom look and feel to your application. The `af:goButton` component skin style selectors are described in Table 20–1.

Table 20–1 af:goButton Component Style Selectors

Name	Description
`af|goButton`	Style on the root element of the `af:goButton` component. You can use any valid CSS-2.1 pseudo-class, like `:hover`, `:active`, or `:focus`, as well as `:disabled`, to style the component for different states. Note that for buttons, the `:active` and `:focus` pseudo-classes do not work in Internet Explorer 7 (IE7). IE7 also does not allow disabled buttons to be styled. You should use the `.AFButton*:alias` selectors as a shortcut to apply a skin to all button components in the same manner.
`af|goButton::icon-style`	Style on the button icon, if the icon attribute is set on the `af:goButton`.
`af|goButton::access-key`	Style on the text of the button. This includes the `.AFButtonAccessKeyStyle:alias` style.

Figure 20–3 shows the application of the default `fusion` skin on the `af:goButton` component and the component icon.

Figure 20–3 af:goButton Component Default Appearance

Figure 20–4 shows the new appearance of the button and icon by setting the following style properties in a custom skin:

```
af|goButton::access-key {color: red;}
af|goButton::icon-style {border: 1px solid black;}
```

Figure 20–4 af:goButton Component with Custom Skin Applied

The ADF Faces skin style selectors used by the default skin are defined in the "Skin Selectors for Fusion's ADF Faces Components" and "Skin Selectors for Fusion's Data Visualization Tools Components" topics in JDeveloper's online help. They are located in **All Online Help** > **Developing Oracle ADF Faces Applications**.

JDeveloper provides coding support while editing your CSS files. You can invoke the CSS code editor when editing your file directly or when editing an ADF Faces component in the JSP source editor. Code support is available for the following:

- Code insight
- Error highlighting
- Preview of styles
- Refactoring
- Finding usages
- Quick comment
- Formatting

- Matching tag highlighting

20.3.1 How to Apply Skins to Text

In addition to using a CSS file to determine styles, skins also use a resource bundle to determine the text within a component. The text that ADF Faces components render can be translated and abstracted as a resource string. For example, `af_chooseDate.LABEL_SELECT_YEAR` is the resource string for the label of the field used to select the year using an `af:chooseDate` component. All the ADF Faces skins use the same resource bundle.

To apply a skin to the text in ADF Faces components, create a custom resource bundle and override the default resource string values. Then, set the `<bundle-name>` property for your custom resource bundle in the `trinidad-skins.xml` file.

Note: ADF Faces components provide automatic translation. The resource bundle used for the components' skin is translated into 28 languages. If a user sets the browser to use the German (Germany) language, any text contained within the components will automatically be displayed in German. For this reason, if you create a resource bundle for a custom skin, you must also create localized versions of that bundle for any other languages the application supports.

See Chapter 21, "Internationalizing and Localizing Pages" for more information.

To create and register a custom resource bundle:

1. In JDeveloper, create a new simple Java class:
   - In the Application Navigator, right-click where you want the file to be placed and choose **New** to open the New Gallery.
   - In the **Categories** tree, select **Java**, and in the **Items** list, select **Java Class**.
   - Enter a name and package for the class. The class must extend `java.util.ListResourceBundle`.
2. Add any keys to your bundle that you wish to override and set the text as needed. Example 20–7 shows the `SkinBundle` custom resource bundle.

Example 20–7 Resource Strings Set in Custom SkinBundle

```
public class SkinBundle extends ListResourceBundle {
        @Override
        public Object[][] getContents() {
          return _CONTENTS;
        }

        static private final Object[][] _CONTENTS = {
          {"af_tableSelectMany.SELECT_COLUMN_HEADER", "Select A Lot"},
          {"af_tableSelectOne.SELECT_COLUMN_HEADER", "Select Just One"},
          {"af_showDetail.DISCLOSED_TIP", "Click to Hide"}
        };
 }
```

3. Set the name of your custom resource bundle in the <bundle-name> parameter of the trinidad-skins.xml file. Example 20–8 shows the custom SkinBundle set in the trinidad-skins.xml file.

Example 20–8 Custom SkinBundle Set in trinidad-skins.xml

```
<skin>
    <id>
       purple.desktop
    </id>
    <family>
       purple
    </family>
    <render-kit-id>
       org.apache.myfaces.trinidad.desktop
    </render-kit-id>
    <style-sheet-name>
       skins/purple/purpleSkin.css
    </style-sheet-name>
    <bundle-name>
       org.apache.myfaces.trinidaddemo.resource.SkinBundle
    </bundle-name>
</skin>
```

Another option for applying skins to text is to use the <translation-source> parameter instead of <bundle-name>. The <translation-source> parameter is an EL binding that points to a Map or a ResourceBundle. The benefit of this option is that you can automatically change the translation value based on any logic that you want at runtime. The <bundle-name> tag takes precedence if both are set. Example 20–9 shows the code for using an EL expression to set the <translation-source> parameter in a bundle map.

Example 20–9 Custom Resource Bundle Map

```
public class SkinTranslationMapDemo
{
  /* Test a skin's translation-source EL pointing to a Map */
  public Map<String, String> getContents()
  {
    return _CONTENTS;
  }

  static private final Map<String, String> _CONTENTS = new HashMap<String,
String>();
  static
  {
    _CONTENTS.put("af_inputDate.LAUNCH_PICKER_TIP", "Launch PickerMap");
    _CONTENTS.put("af_showDetail.DISCLOSED_TIP", "Hide Tip Map");
    _CONTENTS.put("af_showDetail.DISCLOSED", "Hide Map");

  }
}
```

Example 20–10 shows setting the <translation-source> parameter for the resource map in the trinidad-skins.xml file.

Example 20–10 Custom Resource Bundle Map Set in trinidad-skins.xml

```
<skin>
```

```
  <id>
      purple.desktop
  </id>
  <family>
      purple
  </family>
  <render-kit-id>
      org.apache.myfaces.trinidad.desktop
  </render-kit-id>
  <style-sheet-name>
      skins/purple/purpleSkin.css
  </style-sheet-name>
  <translation-source>
      #{skinTranslationMap.resourceBundle}
  </translation-source>
</skin>
```

20.3.2 How to Apply Skins to Icons

You can apply skins to the default icons associated with ADF Faces components by specifying the URL path to the icon image in the icon style selector.

Note that CSS syntax like pseudo-classes (`:hover`, and so forth) and descendant selectors and composite class selectors does not work with icon selectors.

> **Note:** If you are overriding a selector for an icon, use a context-relative path for the URL to the icon image (that is, start with a leading slash (`/`)), and do not use quotation marks.
>
> Also, you must include the width and the height for the icon.

Example 20–11 shows a selector for an icon.

Example 20–11 Selector for an Icon

```
.AFErrorIcon:alias {
      content:url(/adf/images/error.png);
      width:7px; height:18px
 }
```

Icons and buttons can both use the `rtl` pseudo-class. This defines an icon or button for use when the application displays in right-to-left mode. Example 20–12 shows the `rtl` pseudo-class used for an icon.

Example 20–12 Icon Selector Using the rtl Pseudo-Class

```
.AFErrorIcon:alias:rtl {
      content:url(/adf/images/error.png);
      width:16px; height:16px
 }
```

20.3.3 How to Apply Skins to Messages

You can apply style to ADF Faces input components based on whether or not they have certain levels of messages associated with them. When a message of a particular type is added to a component, the styles of that component are automatically modified

to reflect the new status. If styles are not defined for the status in question, then the default styles are used.

In order to define styles for your input components based on message levels that are tied to them, you would append a style pseudo-class to your component definition. For example, to define the base style for the content region of the `af:inputText` component to a background color of purple, use the style selector `af|inputText::content{background-color:purple}`. To define the content region of the component when an error message is present, use the skin style selector `af|inputText:error::content`.

The valid message properties are `:fatal`, `:error`, `:warning`, `:confirmation`, and `:info`.

20.3.4 How to Apply Themes to Components

Themes are a way of implementing a look and feel at a component level. The purpose is to provides a consistent look and feel across multiple components for a portion of a page. A common usage for themes is in a JSF page template where certain areas have a distinct look. For example, a page may have a branding area at the top with a dark background and light text, a navigation component with a lighter background, and a main content area with a light background.

A component that sets a theme exposes that theme to its child components and therefore the theme is inherited. Themes can be set (started or changed) by the following components:

- `af:document`
- `af:decorativeBox`
- `af:panelStretchLayout`
- `af:panelGroupLayout`

The Fusion and BLAF Plus skins (`blafplus-rich` and `blafplus-medium`) support the following themes:

- Dark
- Medium
- Light
- None (default)

In the JSPX page, the theme is started by the `af:document` component, as in:

```
<af:document theme="dark">
  <af:panelTabbed>...</af:panelTabbed>
</af:document>
```

To set the theme for a component, specify a `theme` attribute in the skin selector in the CSS file. For example, the selector to change the text color under an `af:panelTabbed` component to a dark theme is:

```
af|panelTabbed[theme="dark"] {
  color: red;
}
```

If you do not want a child component to inherit modifications made to a parent component in a JSPX page, set a value for the `-tr-children-theme` property in the CSS file. For example, you do not want the `af:panelTabbed` child component to

inherit the dark theme defined for the af:document parent component in the JSPX page. Set the -tr-children-theme property in the CSS file as follows:

```
af|panelTabbed::content {
  -tr-children-theme: default;
}
```

By default, themes are not set for components or their child components. Because themes are inherited, the following values are supported when a component has a theme attribute that is not set:

- not given - If no theme is given, the theme is inherited, as in <af:decorativeBox>...
- #{null}- The theme is inherited; same as not given.
- inherit - The theme is inherited; same as null.
- default - The theme is removed for the component and its child components.
- empty string - If the theme is set to a blank string, it has the same behavior as default. For example, <af:decorativeBox theme=""> will remove the theme for the component and its child components.

Because the themes are added to every HTML element of a component that supports themes and that has style classes, there is no need for containment-style CSS selectors for themes. With the exception of :ltr and :rtl, all theme selectors should always appear on the last element of the selector. For example, the selector to apply a dark theme to each step of an af:breadCrumbs component would be:

```
af|breadCrumbs::step:disabled[theme="dark"] {
  color:#FFFFFF;
}
```

Color incompatibility may occur if a component sets its background color to a color that is not compatible with its encompassing theme color. For example, if a panelHeader component is placed in a dark theme, the CSS styles inside the panelHeader component will set its component background to a light color without changing its foreground color accordingly. The result is a component with a light foreground on a light background. Many other components also set their foreground color to a light color when placed in a dark theme.

If color incompatibility occurs, you can resolve color incompatibility between parent and child components by setting a value for the -tr-children-theme property. For components that do not have a parent-child relationship, you can manually set the component's theme color to a color that is compatible with the surrounding theme color. You do this by inserting the panelGroupLayout or panelStretchLayout component inside the component and by setting the panelGroupLayout or panelStretchLayout theme to a compatible color.

```
<af:panelHeader text="Header Level 0">
            <af:panelGroupLayout layout="vertical" theme="default">
              ...
            </af:panelGroupLayout>
</af:panelHeader>
```

20.3.5 How to Create a Custom Alias

You can create your own alias that you can then include on other selectors.

To create a custom alias:

1. Create a selector class for the alias. For example, you can add an alias to set the color of a link when a mouse cursor hovers over it:

   ```
   .MyLinkHoverColor:alias {color: #CC6633;}
   ```

2. To include the alias in another selector, add a pseudo-element to an existing selector to create a new selector, and then reference the alias using the `-tr-rule-ref:selector` property.

   For example, you can create a new selector for the `af|menuBar::enabled-link` selector to style the hover color, and then reference the custom alias, as shown in Example 20–13.

Example 20–13 Referencing a Custom Alias in a New Selector

```
af|menuBar::enabled-link:hover
{
  -rt-rule-ref:selector(".MyLinkHoverColor:alias");
}
```

20.3.6 How to Configure a Component for Changing Skins Dynamically

To configure a component to dynamically change the skin, you must first configure the component on the JSF page to set a scope value that can later be evaluated by the configuration file. You then configure the skin family in the `trinidad-config` file to be dynamically set by that value.

To conditionally configure a component to set the skin family:

1. Open the main JSF page (such as the `index.jspx` or a similar file) that contains the component that will be used to set the skin family.
2. Configure the page to display the skin family by using the `sessionScope` component.

   Example 20–14 shows an `af:selectOneChoice` component that takes its selected value, and sets it as the value for the `skinFamily` attribute in the `sessionScope` component on the `index.jspx` page.

Example 20–14 Using a Component to Set the Skin Family

```
<af:selectOneChoice label="Choose Skin:" value="#{sessionScope.skinFamily}"
autoSubmit="true">
    <af:selectItem value="blafplus-rich" label="blafplus-rich"/>
    <af:selectItem value="blafplus-medium" label="blafplus-medium"/>
    <af:selectItem value="simple" label="simple"/>
    <af:selectItem value="richDemo" label="richDemo"/>
    <af:selectItem value="mySkin" label="mySkin"/>
</af:selectOneChoice>
```

   The **Refresh** button on the page resubmits the page. Every time the page refreshes, the EL expression is evaluated and if there is a change, the page is redrawn with the new skin.

To conditionally configure a component for changing skins at runtime:

In the `trinidad-config.xml` file, use an EL expression to dynamically evaluate the skin family:

```
<skin-family>#{sessionScope.skinFamily}</skin-family>
```

20.4 Changing the Style Properties of a Component

ADF Faces components use the CSS style properties based on the Cascading Style Sheet (CSS) specification. Cascading style sheets contain rules, composed of selectors and declarations, that define how styles will be applied. These are then interpreted by the browser and override the browser's default settings.

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 8.2.1, "Geometry Management and Component Stretching.".

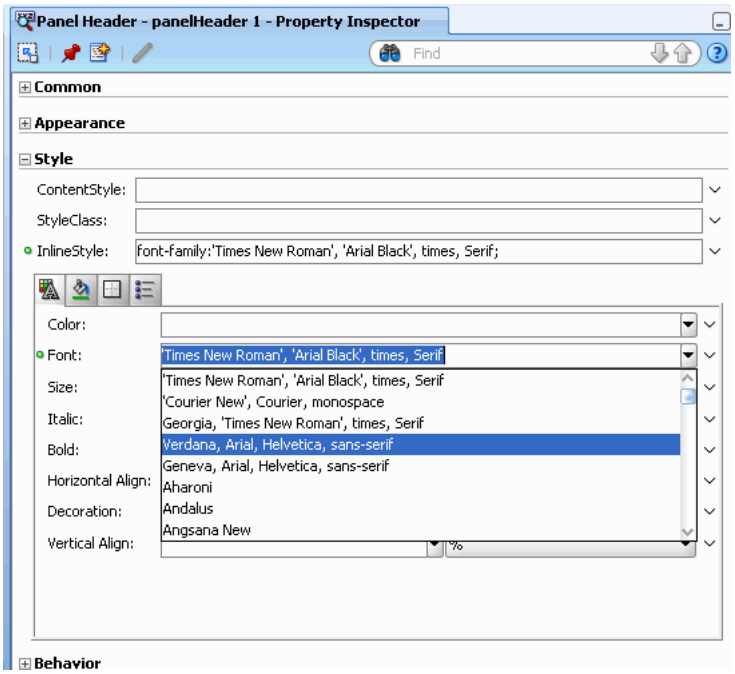
20.4.1 How to Set an Inline Style

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 8.3, "Arranging Contents to Stretch Across a Page".

To set an inline style:

1. Set the `inlineStyle` attribute of the component to the inline style you want to use.
2. If you use the Property Inspector to set a style, you can select the style features you want from dropdown lists, as shown in Figure 20–5.

Figure 20–5 Setting an inlineStyle



JDeveloper adds the corresponding code for the component to the JSF page. Example 20–15 shows the source for an `af:outputText` component with an `inlineStyle` attribute.

Example 20–15 InlineStyle in the Page Source

```
<af:outputText value="outputText1"
               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 20–16.

Example 20–16 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);':''}"/>
```

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 20–17.

Example 20–17 Using the contentStyle Property

```
<af:inputText value="outputText1"
          contentStyle="color:Red;"/>
```

20.4.2 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.

To set a style using a style class:

1. Set the `styleClass` attribute of the component to the style class you want to use.

   Example 20–18 shows an example of a style class being used in the page source.

Example 20–18 Page Source for Using a Style Class

```
<af:outputText value="Text with a style class"
               styleClass="overdue"/>
```

2. 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 20–16.

20.5 Referring to URLs in a Skin's CSS File

You can refer to a URL from a skin's CSS file in a number of different formats. The supported formats are:

- Absolute

You specify the complete URL to the resource. For example, a URL in the following format:

```
http://www.mycompany.com/WebApp/Skin/skin1/img/errorIcon.gif
```

- Relative

  You can specify a relative URL if the URL does not start with `/` and no protocol is present. A relative URL is based on the location of the skin's CSS file. For example, if the skin's CSS file directory is `WebApp/Skin/skin1/` and the specified URL is `img/errorIcon.gif`, the final URL is `/WebApp/Skin/mySkin/img/errorIcon.gif`

- Context relative

  This format of URL is resolved relative to the context root of your web application. You start a context relative root with `/`. For example, if the context relative root of a web application is:

  ```
  /WebApp
  ```

  and the specified URL is:

  ```
  /img/errorIcon.gif
  ```

  the resulting URL is:

  ```
  /WebApp/img/errorIcon.gif
  ```

- Server relative

  A server relative URL is resolved relative to the web server. This differs to the context relative URL in that it allows you reference a resource located in another application on the same web server. You specify the start of the URL using `//`. For example, write a URL in the following format:

  ```
  //WebApp/Skin/mySkin/img/errorIcon.gif
  ```

20.6 Versioning Custom Skins

You can specify version numbers for your custom skins in the `trinidad-skins.xml` file using the `<version>` element. Use this capability if you want to distinguish between custom skins that have the same value for the `<family>` element in the `trinidad-skins.xml` file. Note that when you configure an application to use a particular custom skin, you do so by specifying values in the `trinidad-config.xml` file, as described in Section 20.2, "Applying Custom Skins to Applications."

20.6.1 How to Version a Custom Skin

You specify a version for your custom skin by entering a value for the `<version>` element in the `trinidad-skins.xml` file.

To version a custom skin:

1. In the Application Navigator, double-click the `trinidad-skins.xml` file. By default, this is in the **Web Content/WEB-INF** node.
2. In the structure window, right-click the **skin** node for the custom skin that you want to version and choose **Insert inside skin** > **version**.
3. In the Insert version dialog, select **true** from the default list if you want your application to use this version of the custom skin when no value is specified in the

<skin-version> element of the trinidad-config.xml file, as described in Section 20.2, "Applying Custom Skins to Applications."

4. Enter a value in the name field. For example, enter v1 if this is the first version of the custom skin.
5. Click **OK**.

20.6.2 What Happens When You Version Custom Skins

Example 20–19 shows an example trinidad-skins.xml that references three source files for custom skins (skin1.css, skin2.css, and skin3.css). Each of these custom skins have the same value for the <family> element (test). The values for the child elements of the <version> elements distinguish between each of these custom skins. At runtime, an application that specifies test as the value for the <skin-family> element in the application's trinidad-config.xml file uses skin3 because this custom skin is configured as the default skin in the trinidad-skins.xml file (<default>true</default>). You can override this behavior by specifying a value for the <skin-version> element in the trinidad-config.xml file, as described in Section 20.2, "Applying Custom Skins to Applications."

Example 20–19 trinidad-skins.xml with versioned custom skin files

```
<?xml version="1.0" encoding="windows-1252"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
  <skin>
    <id>skin1.desktop</id>
    <family>test</family>
    <extends>simple.desktop</extends>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <style-sheet-name>skins/skin1/skin1.css</style-sheet-name>
    <version>
      <name>v1</name>
    </version>
  </skin>
  <skin>
    <id>skin2.desktop</id>
    <family>test</family>
    <extends>fusionFx-v1.desktop</extends>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <style-sheet-name>skins/skin2/skin2.css</style-sheet-name>
    <version>
      <name>v2</name>
    </version>
  </skin>
  <skin>
    <id>skin3.desktop</id>
    <family>test</family>
    <extends>fusion.desktop</extends>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <style-sheet-name>skins/skin3/skin3.css</style-sheet-name>
    <version>
     <default>true</default>
      <name>v3</name>
    </version>
  </skin>
</skins>
```

20.7 Deploying a Custom Skin File in a JAR File

You may want to store skin definitions in a Java Archive (JAR) file and then add it to the deployed application. The benefits of packaging skins into a JAR file as compared to bundling them into the application are the following:

- A skin can be deployed and developed separately from the application. This also helps to reduce the number of files to be checked in case some changes must be applied to the skin. Foremost is that using a skin definition contained in a JAR file improves consistency in the look and feel of the application.
- Skin definitions and images can be separated into their own JAR files. Therefore, you can partition the image base into separate JAR files, so that not all files have to be deployed with all applications.

To deploy a skin into a JAR file, follow these rules:

- The `trinidad-skins.xml` file that defines the skin and that references the CSS file must be within the `META-INF` directory.
- All image resources and CSS files must also be under the `META-INF` directory. The images must be in a directory that starts with an `adf` root directory or any directory name that is mapped in the `web.xml` file for the resource servlet, as shown in Example 20–20.
- The JAR file must be placed in the `WEB-INF/lib` directory of the view layer project of the application to deploy (or use a shared library at the application-server level).

Example 20–20 web.xml File with Paths

```
<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>
```

To deploy a skin into a JAR file:

1. Create a directory structure similar to the following:

```
c:\temp\META-INF\adf\oracle\skin\images
        META-INF\skins\fusion.css
        META-INF\trinidad-skins.xml
```

2. Confirm that the directory in the `META-INF` directory starts with `adf`. The `images` directory contains all the images used within the `oracleblaf.css` skin. The CSS reference to the images should have a path similar to this:

```
af|inputColor::launch-icon:rtl {
  content:url(../adf/oracle/skin/images/cfsortl.png);
  width: 12; height: 12;
  left:-7px;
  position:relative;
  right:-7px;
  top:5px;
}
```

Note the two leading periods in front of the image path `../adf/oracle/skin/images/cfsortl.png`. This allows the search for the `META-INF` root to start one directory above the `META-INF/skin` directory in which the CSS is located.

3. Check that the `trinidad-skins.xml` file is located in the `META-INF` directory and that it contains content in a format similar to this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
    <skin>
        <id>richdemo.desktop</id>
        <family>richDemo</family>
        <extends>fusion.desktop</extends>
        <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
        <style-sheet-name>skins/richdemo/richdemo.css</style-sheet-name>
    </skin>
</skins>
```

This example defines the skin as `richdemo.desktop` in the `richDemo` family. The `trinidad-skins.xml` file can have more than one skin definition. The `richdemo.css` file (or your custom CSS file) is referenced from the `style-sheet-name` element.

4. To create the JAR file, issue the following command from the `c:\temp` directory:

```
jar -cvf customSkin.jar META-INF/
```

5. Copy the resulting `customSkin.jar` file to the `WEB-INF/lib` directory of the consuming ADF project. Configure the `trinidad-skins.xml` file located on the `WEB-INF` directory of the ADF project.

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

Because the skin can be discovered at runtime, you do not need to code the skin family name.

Note: The skin definition in the JAR file is not displayed in the JDeveloper visual editor. You may see a message in the log window that the skin family could not be found. You can ignore this message.

21

Internationalizing and Localizing Pages

This chapter describes how to configure JSF pages or an application to display text in the correct language of a user's browser.

This chapter includes the following sections:

21.1 Introduction to Internationalization and Localization of 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 will 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 is displayed in the browser while the table is fetching data during the initial load of data or while the table is being scrolled. 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 more information on skins and resource bundles, see Chapter 20, "Customizing the Appearance Using Styles and Skins".

For any text you add to a component, for example if you define the label of an `af:commandButton` 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 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 Property Inspector, text resources are automatically created in the base resource bundle.

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.

For instance, if the title of this page is My Purchase Requests, instead of having My Purchase Requests as the value for the `title` attribute of the `af:panelPage` component, the value is bound 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 21–1.

Example 21–1 Resource Bundle Element in JSF Configuration File

```
<resource-bundle>
  <var>res</var>
  <base-name>resources.UIResources</base-name>
</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 title attribute of the `af:panelPage` component is then bound to the `myDemo.pageTitle` key in that resource bundle, as shown in Example 21–2.

Example 21–2 Component Text Referencing Resource Bundle

```
<af:panelPage text="#{res['myDemo.pageTitle']}"
```

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 21–3 shows the keys for the myDemo page.

Example 21–3 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 21–4 shows the resource bundle version for the Italian (Italy) locale, `UIResources_it`. Note that there is not an 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 21–4 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
```

21.2 Using Automatic Resource Bundle Integration in JDeveloper

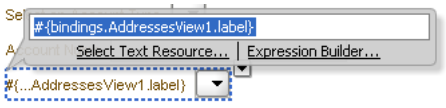
By default, JDeveloper supports the automatic creation of text resources in the default resource bundle when editing ADF Faces components in the visual editor. To treat user-defined strings as static values, disable **Automatically Synchronize Bundle** in the Project Properties dialog, as described in Section 21.2.1, "How to Set Resource Bundle Options".

Automatic resource bundle integration can be configured to support one resource bundle per page or project, or multiple shared bundles.

You can edit translatable text strings using any one of the following methods:

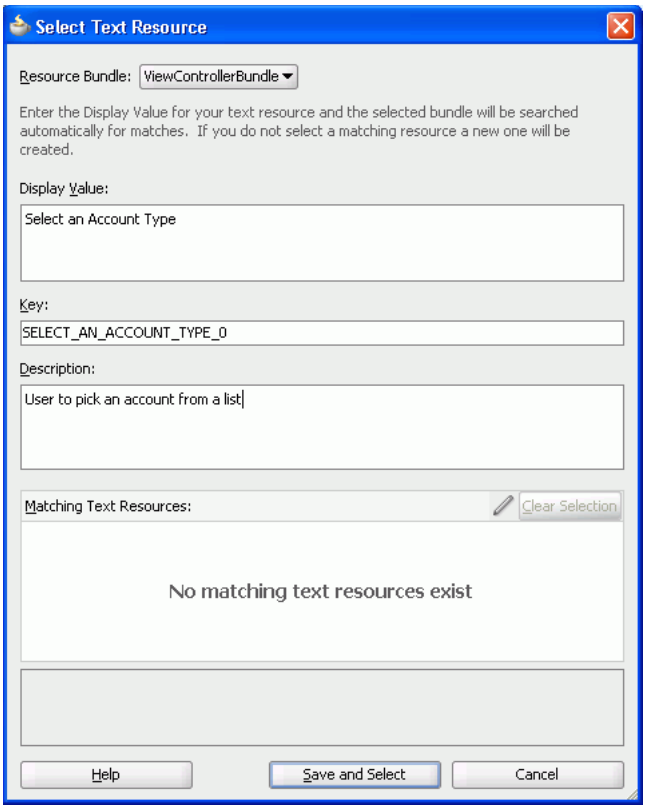
- In the visual editor, enter the new text directly in the component. Click the component to bring up a text input window, as shown in Figure 21–1.

Figure 21–1 Adding Text to a Component



- From the text input window, choose **Select Text Resource** to launch the Select Text Resource dialog, as shown in Figure 21–2. The dialog can also be accessed by right-clicking the component and choosing **Select Text Resource for**, or from the Property Inspector, by clicking the icon to the right of a translatable property and selecting **Select Text Resource**.

Figure 21–2 Select Text Resource Dialog



- From the text input window, select **Expression Builder** to launch the Expression Builder dialog. The dialog can also be accessed from the Property Inspector by clicking the icon to the right of a translatable property and selecting **Expression Builder**.
- In the Property Inspector, enter a valid expression language string for a translatable property.

> **Note:** JDeveloper only writes strings to a resource bundle that you enter using one of the previously-listed methods.

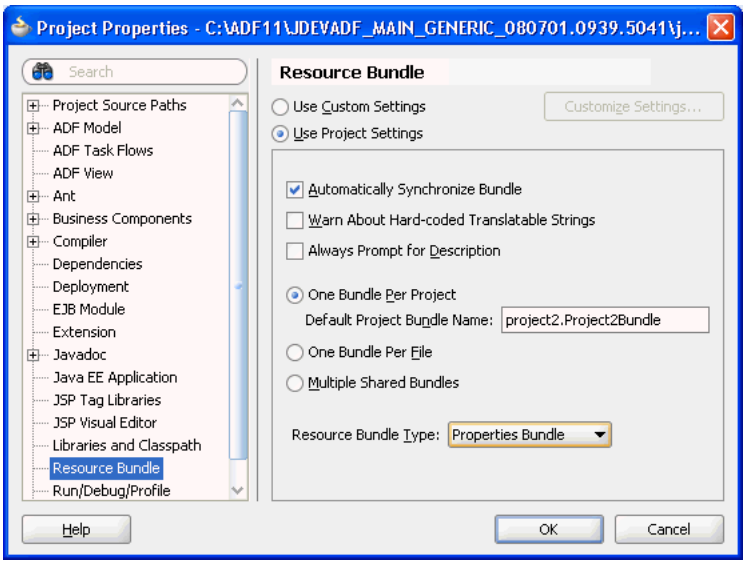
21.2.1 How to Set Resource Bundle Options

After you have created a project, you can set resource bundle options in the Project Properties dialog.

To set resource bundle options for a project:

1. In the Application Navigator, double-click the project.
2. In the Project Properties dialog, select **Resource Bundle** to display the resource bundle options, as shown in Figure 21–3.

Figure 21–3 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 Page** - configured in a file named `<PageName>.properties`.
   - **Multiple Shared Bundles**.
5. Select the resource bundle type from the dropdown list:
   - **XML Localization Interchange File Format (XLIFF) Bundle**
   - **List Resource Bundle**
   - **Properties Bundle**
6. Click **OK**.

21.2.2 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 21–3. It generates a resource bundle the first time that you invoke the Select Text Resource dialog illustrated in Figure 21–2.

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.

21.2.3 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 Property Inspector.

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.

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 Property Inspector, use a property's dropdown list to select **Select Text Resource** to create a new entry in the resource bundle.

   The **Select Text Resource** entry in the dropdown list 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 21–2.

   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 exists.

5. Click **Save and Select**.

21.2.4 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 21.2.2, "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.NAME}
```

where `viewcontrollerBundle` references the resource bundle and `NAME` is the key for the runtime value.

21.3 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` and `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.

21.3.1 How to Define the Base Resource Bundle

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.

To create a resource bundle as a property file or an XLIFF file:

1. In JDeveloper, create a new file.
   - In the Application Navigator, right-click where you want the file to be placed and choose **New** from the context menu to open the New 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.

   - In the **Categories** tree, select **General**, and in the **Items** list, select **File**. Click **OK**.
   - 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 are creating 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.

2. 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 21.3.2, "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 will inherit 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>
```

3. After you have entered all the values, click **OK**.

To create a resource bundle as a Java class:

1. In JDeveloper, create a new Java class:
   - In the Application Navigator, right-click where you want the file to be placed and choose **New** to open the New 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.

   - In the **Categories** tree, select **General**, and in the **Items** list, select **Java Class**. Click **OK**.
   - 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.

2. Implement the `getContents()` method, which simply 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 21.3.2, "How to Edit a Resource Bundle File". Example 21–5 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 21–5 Base Resource Bundle Java Class

```
package sample;

import java.util.ListResourceBundle;

public class MyResources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
}
static final Object[][] contents {
  {"button_Search", Search"},
  {"button_Reset", "Reset"},
  };
}
```

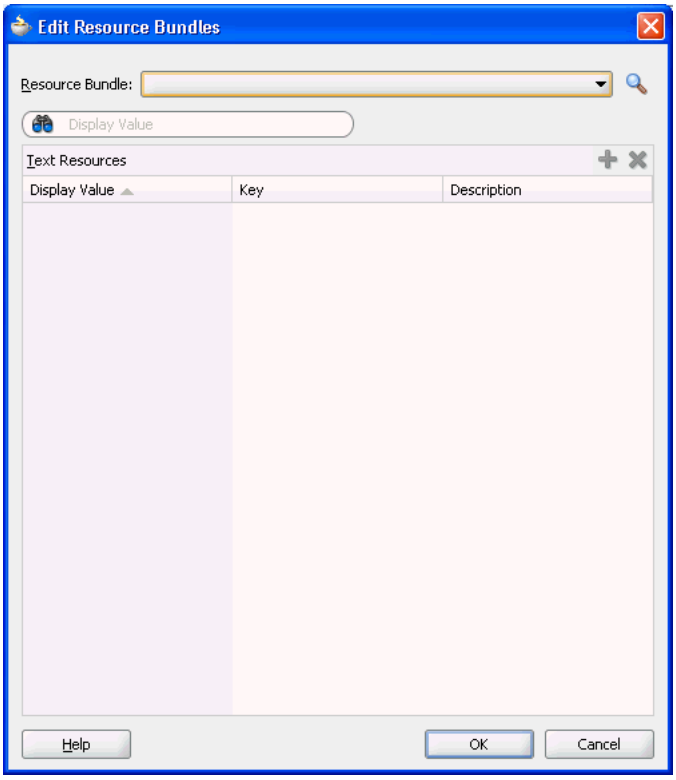
21.3.2 How to Edit a Resource Bundle File

After you have created a resource bundle property file, XLIFF file, or Java class file, you can edit it using the source editor.

To edit a resource bundle after it has been created:

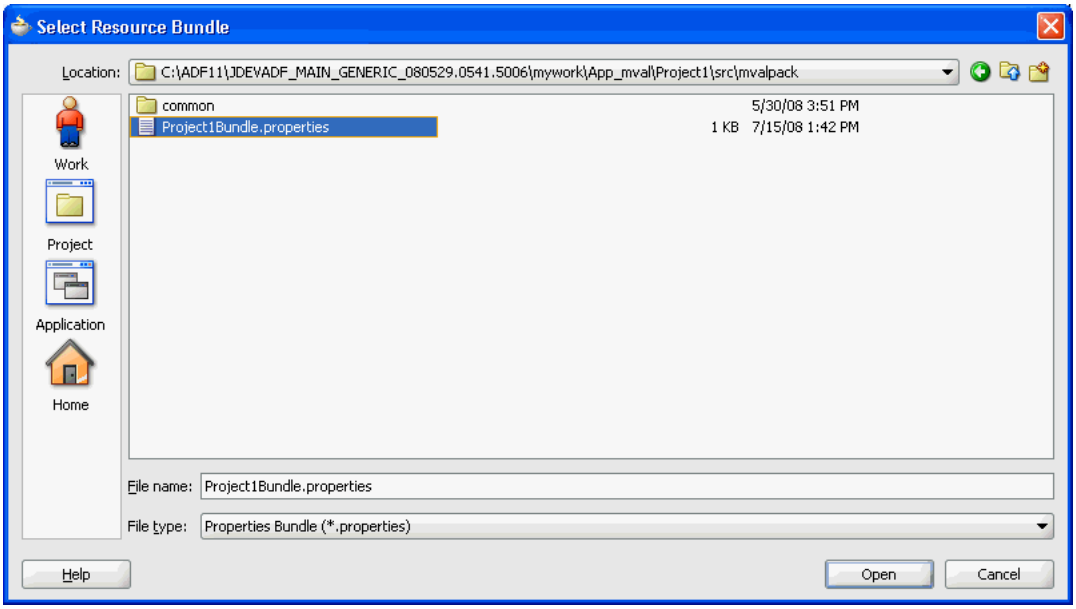
1. In JDeveloper, choose **Application > Edit Resource Bundles** from the main menu.
2. In the Edit Resource Bundles dialog, select the resource bundle file you want to edit from the **Resource Bundle** dropdown list, as shown in Figure 21–4, or click the **Search** icon to launch the Select Resource Bundle dialog.

Figure 21–4 Edit Resource Bundle Dialog



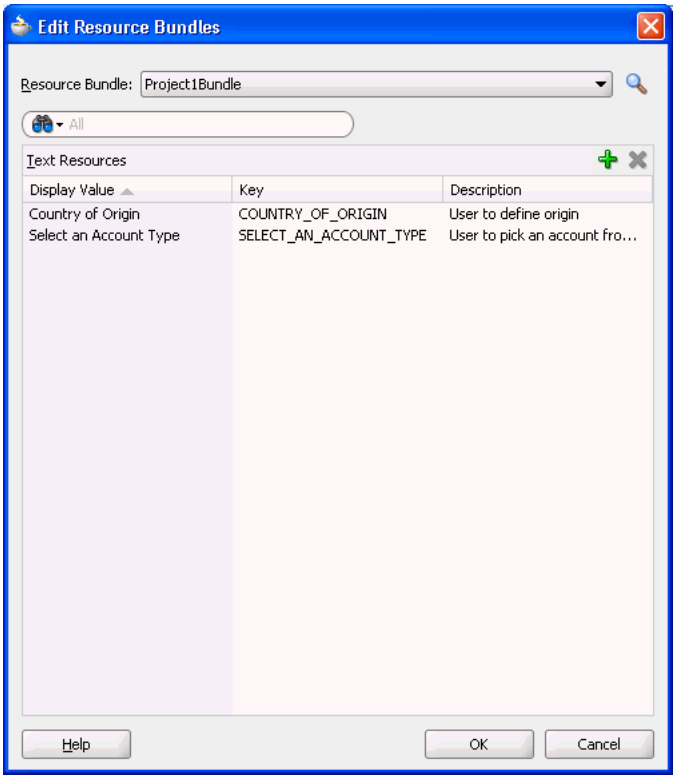
3. In the Select Resource Bundle dialog, select the file type from the **File type** dropdown list. Navigate to the resource bundle you want to edit, as shown in Figure 21–5. Click **OK**.

Figure 21–5 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 21–6. When you have finished, click **OK**.

Figure 21–6 Adding Values to a Resource Bundle



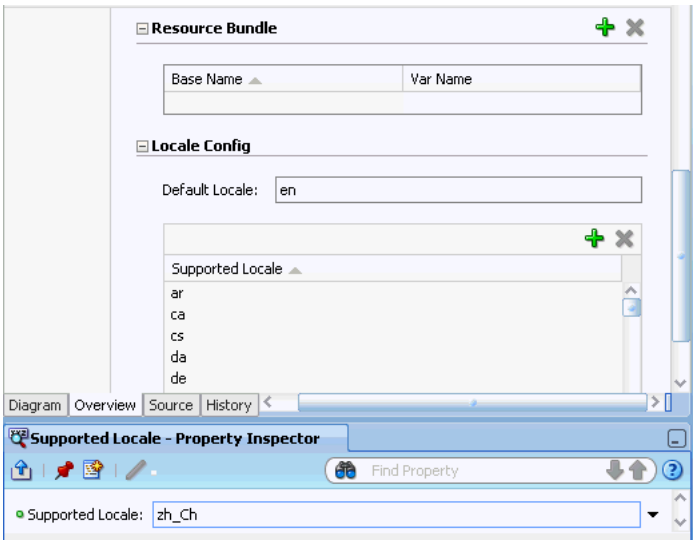
21.3.3 How to Register Locales and Resource Bundles in Your Application

You must register the locales and resource bundles used in your application in the `faces-config.xml` file.

To register a locale for your application:

1. Open the `faces-config.xml` file and click the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
2. In the editor window, select **Application**.
3. In the **Locale Config** area, click **Add** to open the Property Inspector to add the code for the locale, as shown in Figure 21–7.

Figure 21–7 Adding a Locale to faces-config.xml



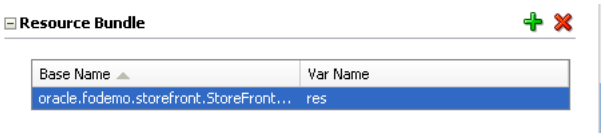
After you have added the locales, the `faces-config.xml` file should have code similar to the following:

```
<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>
```

To register the resource bundle:

1. Open the `faces-config.xml` file and click the **Overview** tab in the editor window. The `faces-config.xml` file is located in the `<View_Project>/WEB-INF` directory.
2. In the editor window, select **Application**.
3. In the **Resource Bundle** section, click **Add** to enable editor input. 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 21–8.

Figure 21–8 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.fodemo.storefront.StoreFrontUIBundle</base-name>
      <var>res</var>
```

```
</resource-bundle>
```

21.3.4 How to Use Resource Bundles in Your Application

With JSF 1.2 you are not required to load the base resource bundle on each page in your application with the `<f:loadBundle>` tag.

To use a base resource bundle on your page:

1. 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 21–6 shows the encoding for a sample page.

Example 21–6 Page and Response Encoding

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:af="http://xmlns.oracle.com/adf/faces"
          xmlns:afc="http://xmlns.oracle.com/adf/faces/webcache">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <f:view>
```

> **Tip:** By default JDeveloper sets the page encoding to `windows-1252`. To set the default to a different page encoding:
>
> 1. From the menu, choose **Tools** > **Preferences**.
> 2. In the left-hand pane, select **Environment** if it is not already selected.
> 3. Set **Encoding** to the preferred default.

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 21–7 shows the code for the **View** button on the myDemo page.

Example 21–7 Binding to a Resource Bundle

```
<af:commandButton 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:commandButton 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 21–8 shows how `adfBundle` is used to provide the button text with a resource strings from a specific resource bundle.

Example 21–8 Binding Using adfBundle

```
<af:commandButton text="#{adfBundle['project.EmpMsgBundle'] ['Deptno_LABEL']}"
```

21.3.5 What You May Need to Know About Custom Skins and Control Hints

If you use a custom 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.

21.3.6 What You May Need to Know About Overriding a Resource Bundle in a Customizable Application

If you are developing a customizable application using the Oracle Metadata Services (MDS) framework and you create a resource bundle (an override bundle) that overrides key-value pairs from the base resource bundle, you need to configure your application's `adf-config.xml` file to support the overriding of the base resource bundle. An override bundle is a resource bundle that contains the key-value pairs that differ from the base resource bundle that you want to use in your customizable application. If, for example, you have a base bundle with the name `oracle.demo.CustAppUIBundle`, you configure an entry in your application's `adf-config.xml` file as shown in Example 21–9 to make it overrideable. Once it is marked as overriden, any customizations of that bundle will be stored in your application's override bundle.

Example 21–9 Entry for Override Bundle in adf-config.xml File

```
<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>
```

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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

21.4 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.

21.4.1 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

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 2.4.1, "How to Create JSF JSP Pages".

2. Provide a control (for example, a `selectOneChoice` component) that an end user can use to change locale.

   For example, in the Components Palette, from the Common Components panel, drag and drop a **Select One Choice** anywhere onto the page.

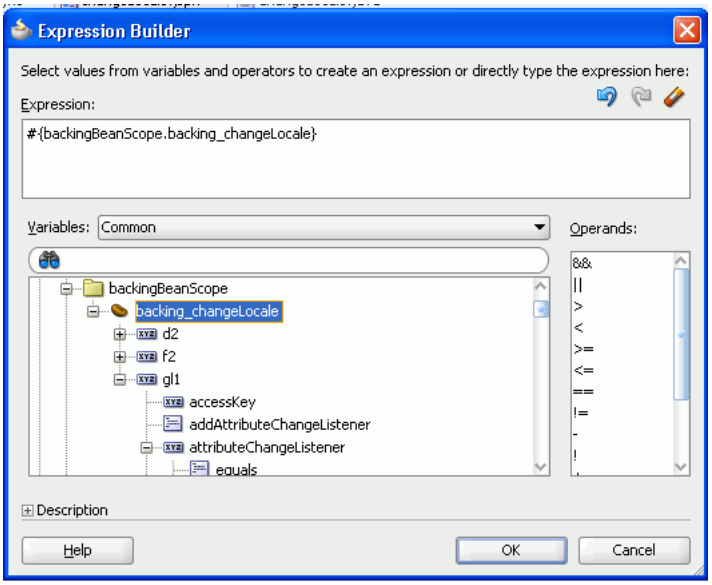
3. Bind the control to a backing bean that stores the locale value, as illustrated in the following example.

```
<af:selectOneChoice label="Select Locale"
    binding="#{backingBeanScope.backing_changeLocale.soc1}"
    id="soc1">
  <af:selectItem label="French" value="FR"
    binding="#{backingBeanScope.backing_changeLocale.si1}"
    id="si1"/>
    ...
</af:selectOneChoice>
```

4. Bind the `locale` attribute of the `f:view` tag to the locale value in the backing bean.

   1. In the Structure window for the JSF page, right-click the `f:view` tag and choose **Go to Properties**.
   2. In the Property Inspector, use the dropdown menu next to the locale attribute to open the Expression Builder.
   3. Use the Expression Builder to bind to the locale value in the backing bean, as shown in Figure 21–9.

Figure 21–9 Expression Builder Binding the Locale Attribute to a Backing Bean



5. Save the page.

21.4.2 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 21–10 shows an example using the `selectOneChoice` component.

Example 21–10 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 21–11 shows extracts for the backing bean that correspond to Example 21–10.

Example 21–11 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;
    }
...
}
```

21.4.3 What Happens at Runtime When 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 21.3.3, "How to Register Locales and Resource Bundles in Your Application".

21.5 Configuring Optional ADF Faces Localization Properties

Along with providing text translation, ADF Faces also automatically provides other types of translation, such as text direction and currency codes. 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 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`.
- `<time-zone>`: Defines the time zone appropriate to the selected locale. ADF Faces automatically uses the time zone used by the client browser. This value is used by `oracle.adf.view.faces.converter.DateTimeConverter` when it converts `String` to `Date`.
- `<formatting-locale>`: Defines the date and number format appropriate to the selected locale. *ADF Faces* and Trinidad, will by default, 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.

21.5.1 How to Configure Optional Localization Properties

You can configure optional localization properties by entering elements in the `trinidad-config.xml` file.

To configure optional localization properties:

1. Open the `trinidad-config.xml` file. The file is located in the `<View_Project>/WEB-INF` directory.
2. From the Component Palette, drag the element you wish to add to the file into the Structure window. An empty element is added to the page.
3. Enter the desired value.

Example 21–12 shows a sample `trinidad-config.xml` file with all the optional localization elements set.

Example 21–12 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>
```

22

Developing Accessible ADF Faces Pages

This chapter describes how to add accessibility support to ADF Faces components with keyboard shortcuts and text descriptions of the component name and state. Accessibility guidelines for ADF pages that use partial page rendering, scripting, styles, and certain page and navigation structures are also described.

This chapter includes the following sections:

22.1 Introduction to Accessible ADF Faces Pages

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.

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`.

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.

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. 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.

While following provided 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.

22.2 Exposing Accessibility Preferences

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, it is recommended to use the `screenReader` mode.
- `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).

It is recommended that you 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 Section 20.1.1, "ADF Faces Skins."

The acceptable values for `<accessibility-profile>` are:

- `high-contrast`: ADF Faces can generate high-contrast–friendly visual content. The 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 high-contrast mode is more beneficial if it is used in conjunction with your browser's or operating system's high-contrast mode. Also, some users might find it beneficial to use the large-font mode along with the high-contrast mode.
- `large-fonts`: ADF Faces can generate browser-zoom–friendly content. In default mode, most text and many containers have a fixed size to provide a consistent and defined look. In the large-font mode, text and containers have a scalable 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 the large-font mode or browser-zoom capabilities, you should disable the large-font mode. Also, some users might find it beneficial to use the high contrast mode along with the 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 preference.

22.2.1 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.

To configure accessibility support in trinidad-config.xml in JDeveloper:

1. In the Application Navigator, double-click **trinidad.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 the 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.
4. Alternatively, you can use the Structure window to insert the value:
   a. In the Application Navigator, select the **trinidad-config.xml** file.
   b. In the Structure window, right-click the XML file root element, choose the **Insert Inside** menu item, and click the **<accessibility-mode>** element.
   c. Double-click the newly inserted element in the Structure window to open the Property Inspector. 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 is returned if the accessibility mode is not explicitly set:

```
<af:outputText value="*#{requestContext.accessibilityMode}"/>
```

22.3 Specifying Component-Level Accessibility Properties

Guidelines for component-specific accessibility are provided in Section 22.3.1, "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 22.5, "Running Accessibility Audit Rules."

Access key support for ADF Faces input or command and go components such as `af:inputText`, `af:commandButton`, and `af:goLink` 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 Property Inspector or in the page source using `&amp` encoding.

Note: Access keys are not displayed if the accessibility mode is set to `screenReader` mode. For more information, see Section 22.2, "Exposing Accessibility Preferences."

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:goButton` and `af:goLink` 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 22.4, "Creating Accessible Pages."

22.3.1 ADF Faces Component Accessibility Guidelines

To develop accessible ADF Faces components, follow the guidelines described in Table 22–1. Components not listed 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 22–1 ADF Faces Components Accessibility Guidelines

Component	Guidelines
`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:commandButton`	One of the following properties must be specified: `text`, `textAndAccessKey`, or `shortDesc`. The text should specify the action to be taken and make sense when read out of context. For example use "go to index" instead of "click here."

Table 22–1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
af:commandLink	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.
af:commandMenuItem af:commandNavigationItem af:comandToolbarButton	One of the following properties must be specified: text, textAndAccessKey, or shortDesc.
af:dialog af:document	Specify the title property.
af:goButton	One of the following properties must be specified: text, textAndAccessKey, or shortDesc. The text should specify the action to be taken and make sense when read out of context. For example use "go to index" instead of "click here."
af:goLink	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.
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 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.
af:outputFormatted	The value property must specify valid HTML.
af:outputLabel	When using this component to label an ADF Faces input or select control, the for property must be specified.
af:panelBox af:panelHeader	Specify the text property.
af:panelLabelAndMessage	When using this component to label an ADF Faces input or select control, the for property must be specified.
af:panelSplitter af:panelStretchLayout	Refer to Section 22.4.4, "How to Use Page Structures and Navigation."

Table 22–1 (Cont.) ADF Faces Components Accessibility Guidelines

Component	Guidelines
af:panelWindow	Specify the title property.
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
af:quickQuery	Specify the searchDesc property.
af:richTextEditor	Specify the label property.
af:selectBooleanCheckbox af:selectBooleanRadio	One of the following properties must be specified: text, textAndAccessKey, or label.
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.
af:showDetailHeader	Specify the text property.
af:showDetailItem	One of the following properties must be specified: text, textAndAccessKey, or shortDesc.
af:table af:treeTable	Specify the summary property. The summary should describe the purpose of the table. If the table is used for layout purposes, the summary property must contain an empty string. All table columns must have column headers.

22.3.2 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.

22.3.3 ADF Data Visualization Components Accessibility Guidelines

To develop accessible ADF Data Visualization components, follow the accessibility guidelines described in Table 22–2. Components not listed do not have accessibility guidelines.

Table 22–2 ADF Data Visualization Components Accessibility Guidelines

Component	Guideline

`dvt:projectGantt`
`dvt:resourceUtilizationGantt`
`dvt:schedulingGantt`
	Specify the `summary` property. The summary should describe the purpose of the Gantt chart component.

`dvt:gauge`	Specify the `shortDesc` property.

Table 22–2 (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 component substitutes 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`	Specify the `summary` property. Note that in screen reader mode, an instance of the tree table component substitutes for the hierarchy viewer component, and the end user can then use the standard cursor keys to navigate through the data.

Table 22–2 (Cont.) ADF Data Visualization Components Accessibility Guidelines

Component	Guideline
`dvt:map`	Specify the `summary` property. Note that in screen reader mode, an instance of the table component substitutes for the map component, and the end user can then use the standard cursor keys to navigate through the data.
`dvt:pivotTable`	Specify the `summary` property. The summary should describe the purpose of the pivot table component.
`dvt:sparkChart`	Specify the `shortDesc` property.

22.3.4 How to Define Access Keys for an ADF Faces Component

In the Property Inspector 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.

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 22–1 shows the code that sets the access key to the letter `h` for the `af:goLink` component. When the user presses the keys ALT+H, the text value of the component will be brought into focus.

Example 22–1 AccessKey Attribute Defined

```
<af:goLink 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 Property Inspector, you need only the `&`.

  Example 22–2 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:commandButton` component.

Example 22–2 TextAndAccessKey Attribute Defined

```
<af:commandButton textAndAccessKey="&amp;Home"/>
```

- `labelAndAccessKey`: Use to simultaneously set the `label` attribute and the access key on an input component, using conventional ampersand notation.

  Example 22–3 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 22–3 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 22–4 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 22–4 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.

22.3.5 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.

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 21.3.1, "How to Define the Base Resource Bundle."
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 21.3.3, "How to Register Locales and Resource Bundles in 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 22–5 shows an entry in a JSF configuration file for a resource bundle.

Example 22–5 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 21, "Internationalizing and Localizing Pages."

22.4 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

22.4.1 How to Use Partial Page Rendering

Screen readers do not reread the full page in a partial page request. Partial page rendering (PPR) causes the screen reader 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 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.

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 the master-detail scenario, some text that indicates that selecting a row on a master component will result in the detail component being updated could be helpful. Alternatively, a help topic that describes the structure of the page and the relationships between components may also be helpful.

22.4.2 How to Use Scripting

Client-side scripting is not recommended for any application problem for which there is a declarative solution and should be kept to a minimum.

Follow these accessibility guidelines when using scripting:

- Do not interact with the component DOM (Document Object Model) 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.

- Provide keyboard equivalents.

  Some users may not have access to a mouse. 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, the function must be accessible in a device-independent way. Practically speaking this means that:

  – All functions must be accessible using the keyboard events.

  – Click events should be preferred over mouseover or mouseout.

  – Mouseover or mouseout events should additionally be available through a click event.

- Avoid focus changes.

  Focus changes can be confusing to screen reader users as these involve a change of context. Applications should avoid changing the focus programmatically, and should never do so in response to focus events. Additionally, popup windows should not be displayed in response to focus changes because standard tabbing will be disrupted.

- Provide explicit popup triggers.

  Screen readers do not automatically respond to inline popup startups. In order to force the screen reader to read the popup contents when in the screen reader mode, the rich client 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.

22.4.3 How to Use Styles

ADF Faces components are already styled and you may not need to make any changes. If you want to use cascading style sheet (CSS) to directly modify their default appearance, you should follow these accessibility guidelines:

- 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.

- Do not use CSS positioning.

  CSS positioning should be used only in the case of positioning the stretched layout component. Do not use CSS positioning elsewhere.

22.4.4 How to Use Page Structures and Navigation

Follow these accessibility guidelines when using these page structures and navigation tools:

- Use `af:panelSplitter` component for layouts.

  When implementing geometry-managed layouts, using `af:panelSplitter` 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 `af:panelStretchLayout`, you should consider using `af:panelStretchLayout` instead when is 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 `first` facet of a vertical `af:panelSplitter` component, rather than within the `top` facet of `af:panelStretchLayout` 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.

- Enable scrolling of flow layout contents.

  When nesting flow layout contents such as layout controls inside of geometry-managed parent components such as `af:panelSplitter` or `af:panelStretchLayout`, wrap `af:panelGroupLayout` with `layout="scroll"` 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.

- Use header-based components to identify page structure.

  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:

  – `af:panelHeader`

  – `af:showDetailHeader`

  – `af:showDetailItem` in `af:panelAccordion` (each accordion in a pane renders an HTML header for the title area)

- Use `af:breadCrumbs` component to identify page location.

Accessibility standards require that users be able to determine their location within a web site or application. The use of `af:breadCrumbs` achieves this purpose.

22.4.5 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 22–3.

Table 22–3 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.

22.5 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 22–1, " ADF Faces Components Accessibility Guidelines." Running an audit report involves creating and running an audit profile.

To create an audit profile:

1. From the main menu, choose **Tools** > **Preferences**.

2. In the Audit: Profiles dialog, deselect all checkboxes except **ADF Faces Accessibility Rules**.
3. Save the profile with a unique name and click **OK**.

To run the audit report:

1. From the main menu, choose **Build** > **Audit** ***target***.
2. Select the audit profile you created from the list.
3. Click **OK** to generate the report.

The audit report results are displayed in the Log window. After the report completes, you can export the results to HTML by clicking the **Export** icon in the Log window toolbar.

Part IV

Using ADF Data Visualization Components

Part IV contains the following chapters:

23

Introduction to ADF Data Visualization Components

This chapter highlights the common characteristics and focus of the ADF Data Visualization components, which are an expressive set of interactive ADF Faces components. 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:

23.1 Introduction to 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, Property Inspector, and Component Palette.
- 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.

For information about the data binding of ADF Data Visualization Components, see the "Creating Databound ADF Data Visualization Components" chapter in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

23.2 Defining the ADF Data Visualization Components

The ADF Data Visualization components include the following: graph, gauge, pivot table, geographic map, Gantt chart, and hierarchy viewer.

23.2.1 Graph

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:

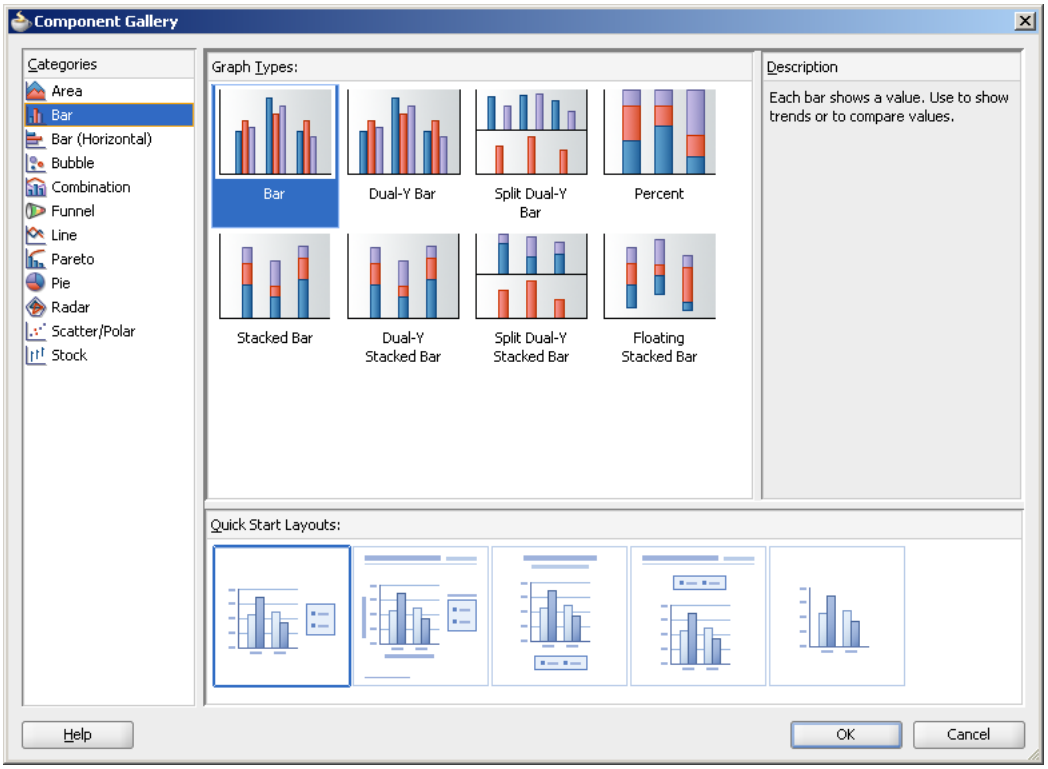
- Area graph: Creates a graph in which data is represented as a filled-in area. Use area graphs to show trends over time, such as sales for the last 12 months. Area graphs require at least two groups of data along an axis. The axis is often labeled with increments of time such as months.
- Bar graph: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: Creates a graph in which data shows the high, low, and closing prices of a stock. Each stock marker displays three separate values.

In JDeveloper, you can 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 23–1 shows the Component Gallery that displays when creating a graph from a data control.

For information about the data binding of graphs, see the "Creating Databound ADF Graphs" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

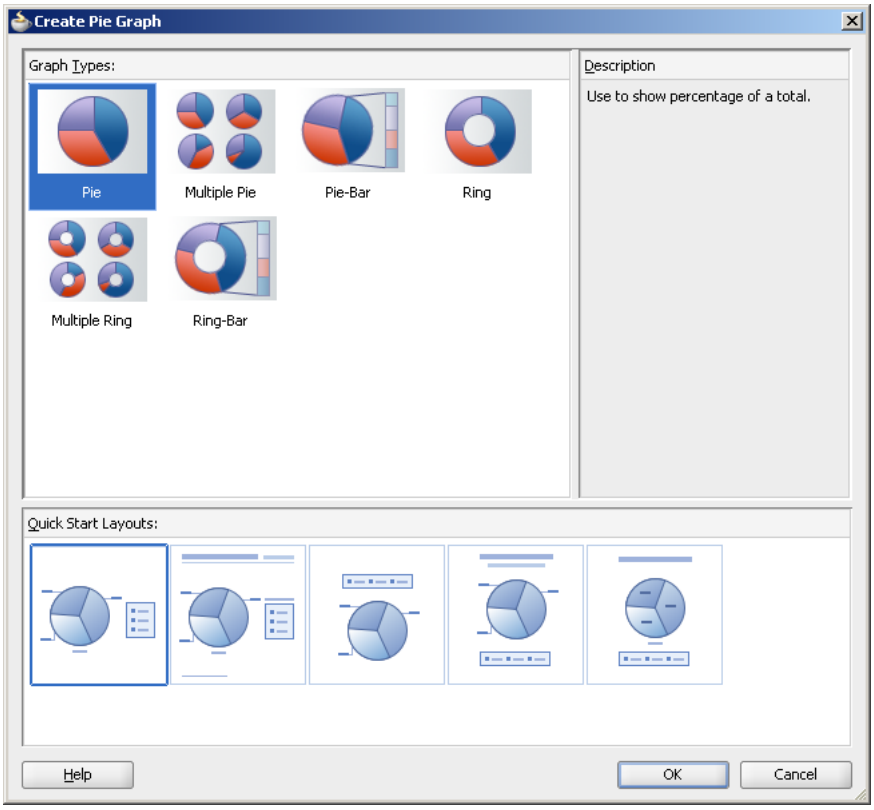
Figure 23–1 Component Gallery for Graphs from Data Controls Panel



You can also create a graph on your page by dragging a graph component from the Component Palette. This approach allows you the option of designing the graph user interface before binding the component to data. Figure 23–2 shows the Component Gallery that displays when creating a pie graph from the Component Palette.

Note: The sparkchart component can only be inserted from the Component Palette and bound to data afterwards.

Figure 23–2 Component Gallery for Pie Graphs from Component Palette



All graphs support HTML5, Flash, SVG, and PNG rendering. Graph components support interactivity on initial display and data change including the use of zooming and scrolling, the use of an adjustable time selector window to highlight specific sections on a time axis, the use of line and legend highlighting and fading to filter the display of data points, and the use of dynamic reference lines and areas.

Figure 23–3 show an application dashboard that illustrates:

- bar graph
- pie graph with exploded slice

Figure 23–3 Dashboard with Bar Graph and Pie Graph

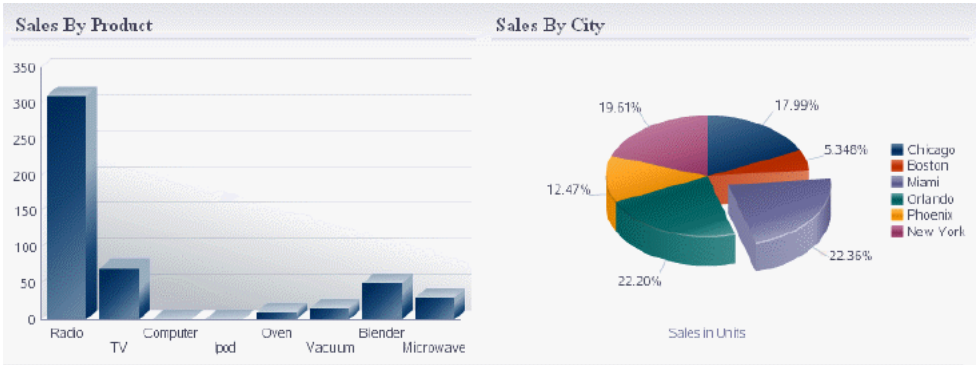


Figure 23–4 shows an application dashboard that illustrates, clockwise from top left:

- curved line graph with time selector window

- pie graph with 3D effect and an exploded slice
- bar graph with 3D effect

Figure 23–4 Dashboard with Line, Pie, and Bar Graphs

Figure 23–5 shows a line sparkchart displaying sales trends in a table column.

Figure 23–5 Sparkchart of Sales Trends

Product	Sales by Month 2008	December Sales (Millions)
HDTV		0
DVD Player		23
Laptop Computer		9
Playstation 3		16
iPod		4
iPhone		19
Blackberry		63
Microwave		28

23.2.2 Gauge

The 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.

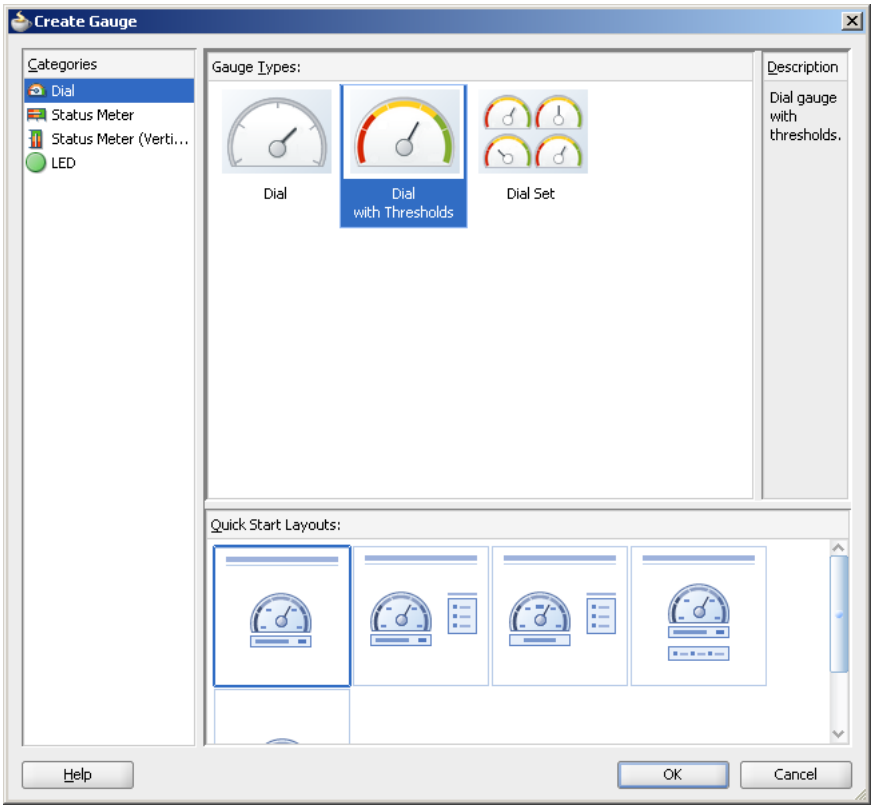
The following kinds of gauges can be produced by this component:

- 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.

In JDeveloper, a Component Gallery displays available gauges categories, types, and descriptions to provide visual assistance when designing gauges and defining a quick layout. Figure 23–6 shows the Component Gallery for gauges.

Figure 23–6 Component Gallery for Gauges



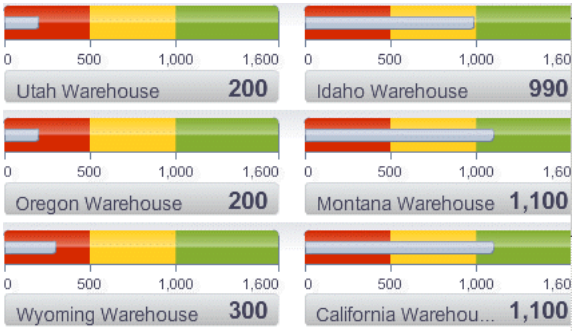
All gauge components can use HTML5, Flash, SVG, and PNG rendering.

Figure 23–7 shows a set of dial gauges set with thresholds to display warehouse stock levels.

Figure 23–7 Dial Gauges set with Thresholds

Figure 23–8 shows a set of status meter gauges set with thresholds.

Figure 23–8 Status Meter Gauges set with Thresholds



23.2.3 Pivot Table

The pivot table produces a grid that supports multiple layers of data labels on rows or columns. An optional pivot filter bar 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. In JDeveloper, a **Create Pivot Table** wizard provides declarative support for databinding and configuring the pivot table. For more information, see the "Creating Databound ADF Pivot Tables" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

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 23–9 shows an example pivot table with a pivot filter bar.

Figure 23–9 Pivot Table with Pivot Filter Bar

Filters: Direct

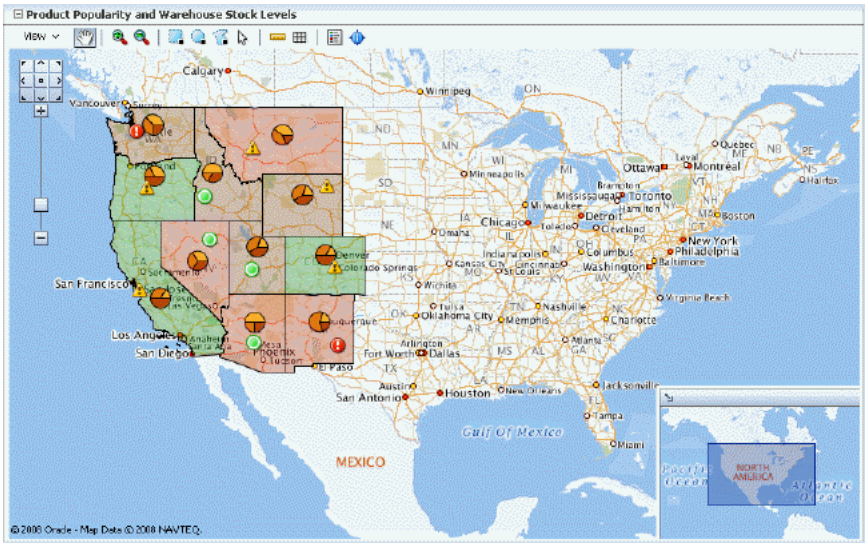
		World		Boston		Total Geography	
		Sales	Units	Sales	Units	Sales	Units
▽ 2005		8,750	35	500	9	9,250	44
	Canoes	3,750	20	375	2	4,125	22
	Tents	5,000	50	125	15	5,125	65
▽ 2006		17,500	70	1,000	15	18,500	85
	Canoes	7,500	40	750	4	8,250	44
	Tents	10,000	100	250	25	10,250	125
▽ 2007		15,000	28	900	11	15,900	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

23.2.4 Geographic Map

The geographic map provides the functionality of Oracle Spatial within the ADF framework. 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 23–10 shows a geographic map with color theme, pie graph theme, and point theme.

Figure 23–10 Geographic Map with Color Theme, Pie Graph Theme, and Point Theme



23.2.5 Gantt Chart

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.

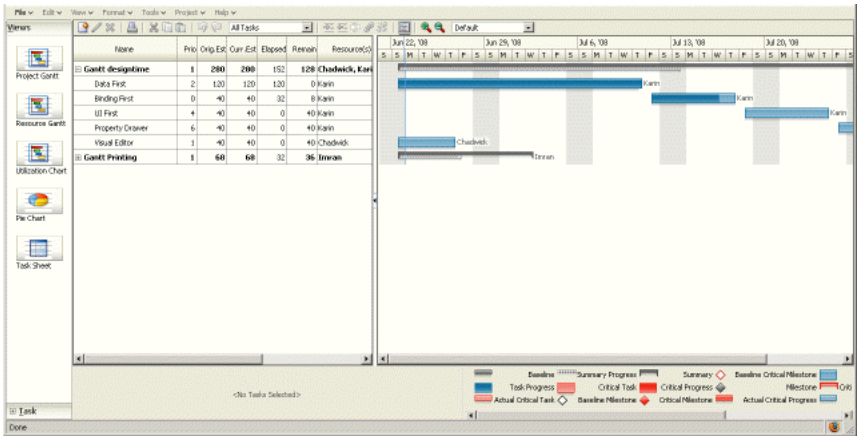
A Gantt chart consists of two ADF Faces tree tables combined with a splitter. The left-hand table contains a list of tasks or resources while the right-hand table consists of a single column in which progress is graphed over time.

There are three types of gantt components:

- Project Gantt: 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: 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: 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 23–11 shows a project Gantt view of staff resources and schedules.

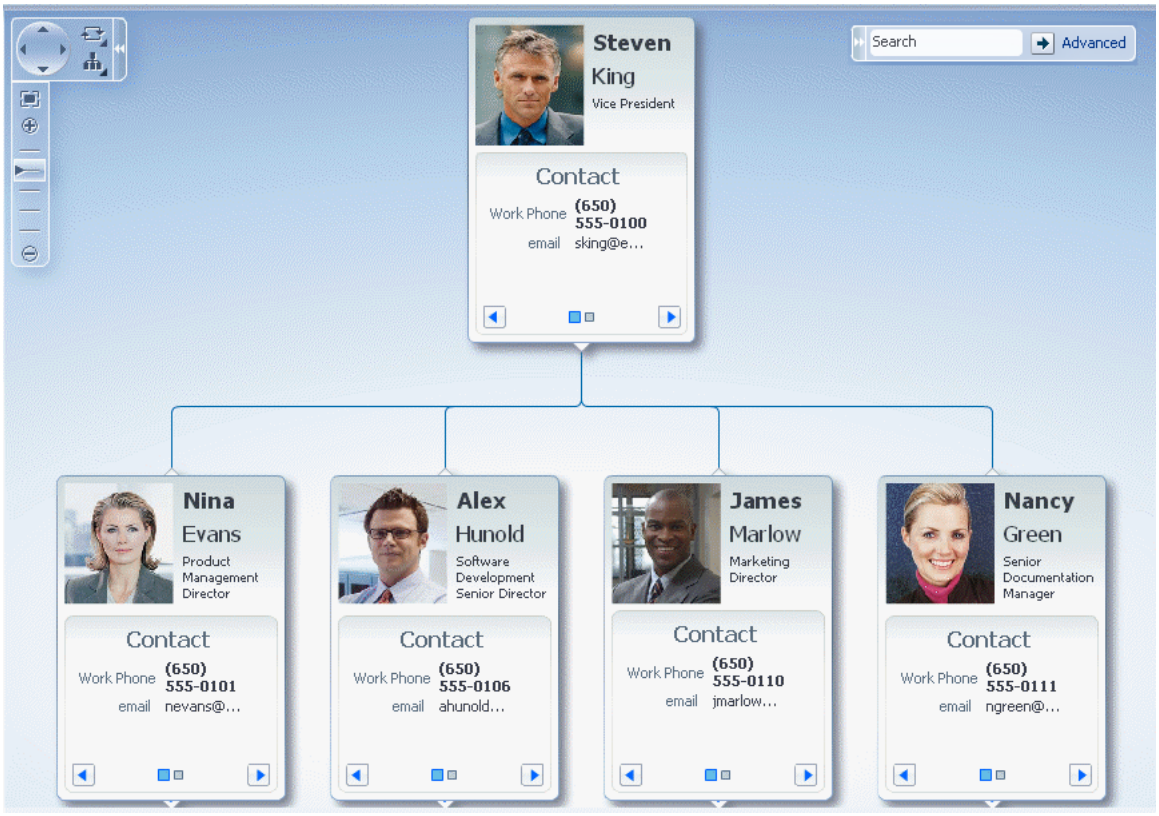
Figure 23–11 Project Gantt



23.2.6 Hierarchy Viewer

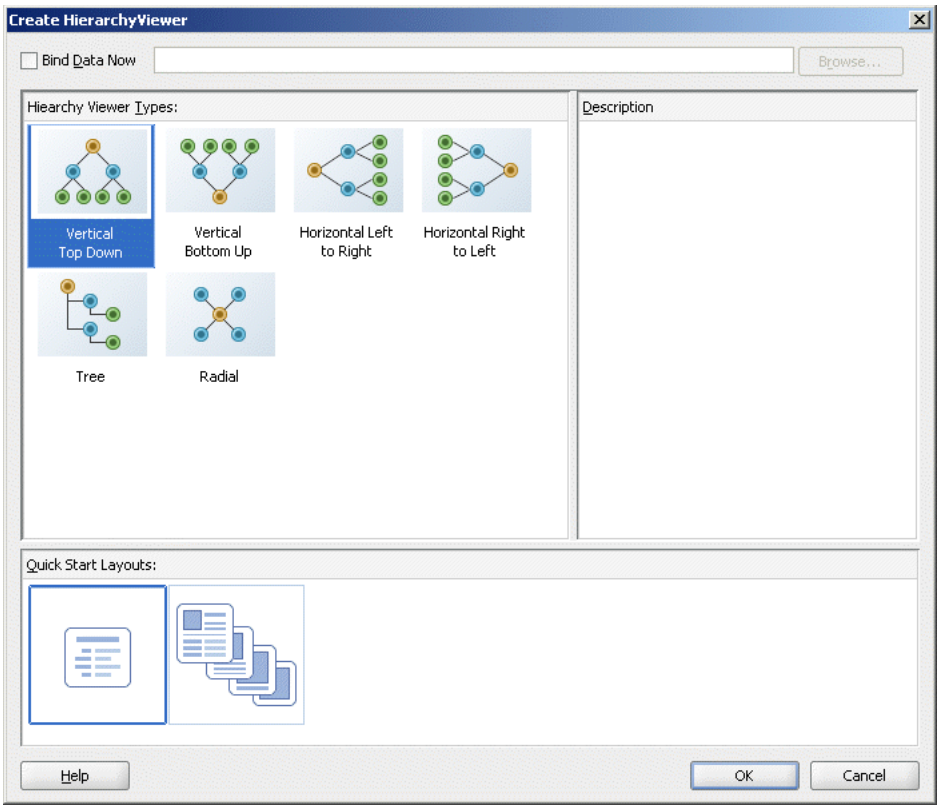
The hierarchy viewer component displays hierarchical data as a set of linked nodes in a diagram. The nodes and links correspond to the elements and relationships to the data. The component supports pan and zoom operations, expanding and collapsing of the nodes, rendering of simple ADF Faces components 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 23–12.

Figure 23–12 Hierarchy Viewer as Organizational Chart



In JDeveloper, a Component Gallery displays available hierarchy viewer types and descriptions to provide visual assistance when designing the component and defining a quick layout. Figure 23–13 shows the Component Gallery for the hierarchy viewer.

Figure 23–13 Component Gallery for Hierarchy Viewer



23.3 Providing Data for ADF Data Visualization Components

All data visualization components can be bound to row set data collections in an ADF data control. For information and examples of data binding these components to data controls, see the "Creating Databound ADF Data Visualization Components" chapter in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Graphs and gauges have a `tabularData` method that lets you provide CSV (Comma Separated Value) data from a method that is stored in a managed bean.

The Gantt chart component supports the use of a basic tree data control when you want to provide data not only for tasks and resources but also for subtasks and subresources.

23.4 Downloading Custom Fonts for Flash Images

Graph and gauge components provide text rotation, high fidelity display, and embedded fonts using Flash image types. The Flash engine is a prebuilt Shockwave Flash (SWF) file containing precompiled `ActionScript` code used to display a graph or gauge by using an XML definition of a chart. The Flash engine is downloaded and instantiated by a Flash Player embedded in the client browser at runtime.

Embedded fonts are used for display and printing purposes, they are not installed on the client, and they cannot be edited. They are used by the Flash Player, in memory, and are cleared when the player terminates. Although embedded fonts require a roundtrip to the server to download the font SWF file, they provide a consistent look across all clients, support text rotation, and minimize distortion or anti-aliasing.

Oracle provides one font, Albany WT, for use in Flash images when necessary. This font does not provide any non-plain variations such as Bold or Italic. The Albany WT font is used instead of the default font to support certain animations not supported by Flash with device fonts, if the application does not specify and provide its own embedded font to use instead.

Specific fonts and their respective SWF files can be added to your application as embedded fonts to be passed to the Flash engine. The engine will defer-load any font specified in the list until that font is required by any text or labels in a graph or gauge definition. Example 23–1 defines the Georgia font with a Bold and Italic combination.

Example 23–1 SWF File

```
package
{
  import.flash.display.Sprite;
  import.flash.text.Font;
public class fGeorgiaBoldItalic extends Srite
(
  [Embed (source="c:\\WINDOWS\\Fonts\\GEORGIABI.TTF",
    fontName="Georgia Bold Italic",
    fontWeight="Bold",
    fontStyle="Italic".
    mimType="application/x-font-truetype")]
  private statis car font1:Class;
  public function fGeorgiaBoldItalic() {
    Font registerFont(font1);
}
  }
}
```

You can set graph and gauge font attributes as follows:

- `fontEmbedding`: Defines whether or not the embedded fonts are used. Some performance may be gained by setting the attribute to `none`.
- `fontMap`: Contains the actual map of the fonts that should be used for embedding. The map contains the name of a font and a URL where the custom font SWF file can be found.

24

Using ADF Graph Components

This chapter describes how to use an ADF graph component to display data and provides the options for graph customization.

This chapter includes the following sections:

- Section 24.1, "Introduction to the Graph Component"
- Section 24.2, "Understanding the Graph Tags"
- Section 24.3, "Understanding Data Requirements for Graphs"
- Section 24.4, "Creating a Graph"
- Section 24.5, "Changing the Graph Type"
- Section 24.6, "Customizing the Appearance of Graphs"
- Section 24.7, "Customizing the Appearance of Specific Graph Types"
- Section 24.8, "Adding Specialized Features to Graphs"
- Section 24.9, "Animating Graphs"

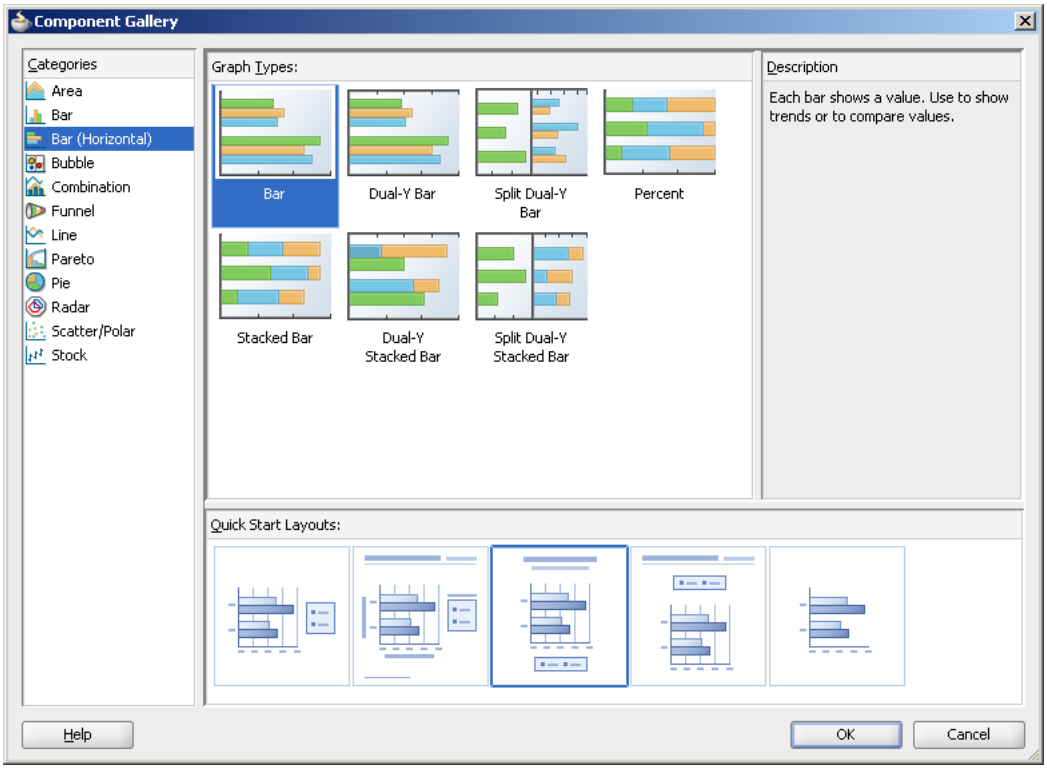
For information about the data binding of ADF graphs, see the "Creating Databound ADF Graphs" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

24.1 Introduction to 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 Component Gallery displays available graph categories, types, and descriptions to provide visual assistance when you are creating graphs and specifying a quick-start layout. Figure 24–1 shows the Component Gallery for horizontal bar graphs.

Figure 24–1 Component Gallery for Horizontal Bar Graphs



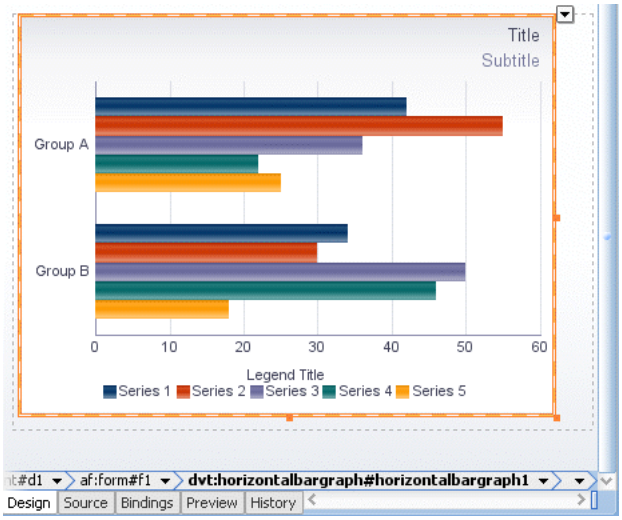
When a graph is inserted into a JSF page using the Component Gallery, a set of child tags that support customization of the graph is automatically inserted. Example 24–1 shows the source code for a horizontal bar graph with the quick-start layout selected in the Component Gallery in Figure 24–1.

Example 24–1 Horizontal Bar Graph Sample Code

```
<dvt:horizontalBarGraph id="horizontalBarGraph1"
        value="#{bindings.SalesStageDataView1.graphModel}"
        subType="BAR_HORIZ_CLUST">
        customLayout="CL_NONE"
  <dvt:background>
    <dvt:specialEffects/>
  </dvt:background>
  <dvt:graphPlotArea/>
  <dvt:seriesSet>
      <dvt:series/>
  </dvt:seriesSet>
  <dvt:o1Axis/>
  <dvt:y1Axis/>
  <dvt:legendArea automaticPlacement="AP_NEVER" position="LAP_BOTTOM"/>
  <dvt:legendTitle text="Legend Title"/>
  <dvt:graphSubtitle horizontalAlignment="CENTER" text="Subtitle"/>
  <dvt:graphTitle horizontalAlignment="CENTER" text="Title"/>
</dvt:horizontalBarGraph>
```

Figure 24–2 shows the visual editor display of the horizontal bar graph created with the Component Gallery in Figure 24–1.

Figure 24–2 Horizontal Bar Graph in Visual Editor



When editing a graph in the visual editor, graph components such as the title, legend area, plot area, background, axis labels, and display of bars can be selected to display a context menu with editing choices. For more information about editing a graph in the visual editor, see Section 24.6, "Customizing the Appearance of Graphs."

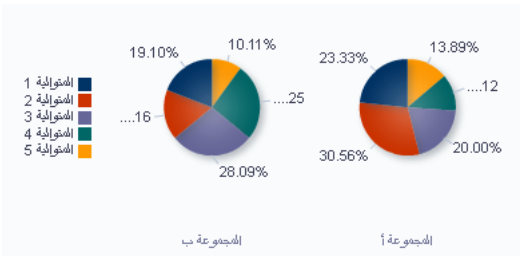
Graphs are displayed in a default size of 400 X 300 pixels. You can customize the size of a graph or specify dynamic resizing to fit an area across different browser window sizes. When graphs are displayed in a horizontally or vertically restricted area, for example in a web page sidebar, the graph is displayed in a fully featured, although, simplified display.

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 22.2, "Exposing Accessibility Preferences." For information about ADF pivot tables, see Section 26.1, "Introduction to the ADF Pivot Table Component."

By default, graphs are displayed in the Flash image format. Alternatively, graphs can be displayed using HTML5 or a Portable Network Graphics (PNG) output format. For more information about graph image formats, see Section 24.4.3, "What You May Need to Know About Graph Image Formats."

HTML5, Flash, and PNG image formats for graphs support bi-directional locales. Figure 24–3 shows bi-directional support in multiple pie graphs displayed with a Flash image format.

Figure 24–3 Bi-directional Support in Graphs



24.2 Understanding the Graph Tags

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-specific tags: The 13 graph-specific tags provide a convenient and quick way to create one commonly used graph type. They are represented in the Component Gallery as categories of graphs with one or more types, and a variety of quick-start layout options from which to choose. For a list and description of these tags, see Section 24.2.1, "Graph-Specific Tags."
- Common graph child tags: These tags are supported by most graph types to provide customization. For a list and description of these tags, see Section 24.2.2, "Common Graph Child Tags."
- Graph-type child tags: These tags apply either to specific graph types or to specific parts of a graph. For a list and description of these tags, see Section 24.2.3, "Graph-Specific Child Tags."
- Child set tags: These tags wrap a set of an unlimited number of related tags. For a list and description of these tags, see Section 24.2.4, "Child Set Tags."
- Graph tag: Although available, but not the preferred method, the `dvt:graph` tag lets you create an instance of a graph for any supported graph type. Even though some graph attributes and some graph child tags are meaningful only for certain graph types, the graph tag has a complete set of graph attributes and supports the use of all graph child tags. Therefore, the `dvt:graph` tag provides full flexibility for choosing graph types, customizing graphs, and changing from one graph type to another.

> **Note:** In JDeveloper the `dvt:graph` tag is not available to declaratively insert from the Component Palette. The tag and its child tags must be typed into the source code of a `.jspx` file.

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 tag in JDeveloper, select the tag in the Structure window and press **F1**. To access the full ADF Data Visualization Tools tag library in JDeveloper Help, expand the Javadoc and Tag Library References node in the online Help Table of Contents and click the link to the tag library in the JDeveloper Tag Library Reference topic.

24.2.1 Graph-Specific Tags

There are 13 graph-specific tags:

- `dvt:areaGraph`: Supports an area 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.
- `dvt:barGraph`: Supports a bar graph in which data is represented 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.

- `dvt:horizontalBarGraph`: Creates a graph that displays bars horizontally along the y-axis. Use horizontal bar graphs to provide an orientation that allows you to show trends or compare values.
- `dvt:bubbleGraph`: Creates a graph in which data is represented 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.
- `dvt:comboGraph`: Creates a 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.
- `dvt: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 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.
- `dvt: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 the Americas, Europe, and Asia. Use line graphs to compare items over the same time.
- `dvt: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 Pareto graphs to identify and compare the sources of defects.
- `dvt: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 pie graphs to show the relationship of parts to a whole such as how much revenue comes from each product line.
- `dvt:radarGraph`: Creates 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.
- `dvt:scatterGraph`: Creates a graph in which data is represented 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.
- `dvt:sparkChart`: Creates 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.

> **Note:** In this release sparkcharts are created by inserting the `dvt:sparkChart` tag from the Component Palette and then binding the component to data. Sparkcharts cannot be created using the Data Controls panel.

- `dvt:stockGraph`: Creates a graph in which data shows 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.

24.2.2 Common Graph Child Tags

Types of common customization and related child tags include:

- Animation effects for graphs: `dvt:animationOnDisplay` and dvt:`animationOnDataChange` tags.
- Alerts that highlight a data point with a custom icon: `dvt:alertSet` and `dvt:alert` tags.
- Annotations that insert notes for specific data points: `dvt:annotationSet` and `dvt:annotation` tags.
- Appearance and titles for the graph: `dvt:background`, `dvt:graphFont`, `dvt:graphFootnote`, `dvt:graphPlotArea`, `dvt:graphSubtitle`, and `dvt:graphTitle` tags.
- Colors and appearance of bars, areas, lines, and pie slices (also known as series items): `dvt:seriesSet` and `dvt:series` tags.
- Formatting categorical attributes in the ordinal axis and marker tooltips: `dvt:attributeFormat`.
- Legend appearance: `dvt:legendArea`, `dvt:legendText`, and `dvt:legendTitle` tags.
- Marker customization related to each axis: `dvt:markerText`, `dvt:x1Format`, `dvt:y1Format`, `dvt:y2Format`, and `dvt:zFormat` tags.
- Reference lines and reference areas: `dvt:referenceObjectSet` and `dvt:referenceObject` tags.
- Customization for the ordinal axis (also known as the category axis) used with bar, area, combination, line, radar, and stock graphs with group labels: `dvt:o1Axis`, `dvt:o1MajorTick`, `dvt:o1TickLabel`, and `dvt:o1Title` tags.
- Customization for the x-axis used with scatter and bubble graphs with numerical labels: `dvt:x1Axis`, `dvt:x1MajorTick`, `dvt:x1TickLabel`, `dvt:x1MinorTick`, and `dvt:x1Title` tags.
- Customization for the y1-axis: `dvt:y1Axis`, `dvt:y1BaseLine`, `dvt:y1MajorTick`, `dvt:y1TickLabel`, `dvt:y1MinorTick`, and `dvt:y1Title` tags.
- Customization for the y2-axis: `dvt:y2Axis`, `dvt:y2BaseLine`, `dvt:y2MajorTick`, `dvt:y2TickLabel`, `dvt:y2MinorTick`, and `dvt:y2Title` tags.

24.2.3 Graph-Specific Child Tags

Types of graph-specific customizations and related child tags include:

- 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` subcomponents: `dvt:specialEffects` and `dvt:gradientStopStyle` tags.
- Interactivity specifications for subcomponents of a graph: `dvt:shapeAttrbutesSet` and `dvt:shapeAttributes` tags.
- Formatting numerical data values for graph: `dvt:sliceLabel`, `dvt:x1TickLabel`, `dvt:y1TickLabel`, `dvt:y2TickLabel`, `dvt:x1Format`, `dvt:y1Format`, `dvt:y2Format`, `dvt:zFormat`, and `dvt:stockVolumeFormat`.
- Time axis customization for area, bar, combination, line, and stacked bar graphs: `dvt:timeAxisDateFormat`, and `dvt:timeSelector` tags.
- Selection of a range on a time axis for master-detail graphs: `dvt:timeSelector` tag.
- Pareto graphs: `dvt:paretoLine` and `dvt:paretoMarker` tags.
- Pie graphs: `dvt:graphPieFrame`, `dvt:pieFeeler`, `dvt:slice`, and `dvt:sliceLabel` tags.
- Sparkcharts: `dvt:sparkItem` tag provides data for the sparkchart.
- Stock graphs: `dvt:stockMarker`, `dvt:stockVolumeformat`, and `dvt:volumeMarker` tags.

24.2.4 Child Set Tags

Child set tags include:

- `dvt:alertSet` tag: 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.
- `dvt:annotationSet` tag: Wraps `dvt:annotation` tags that define an annotation on a graph. An annotation is associated with a specific data point on a graph
- `dvt:referenceObjectSet` tag: 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.
- `dvt:seriesSet` tag: Wraps `dvt:series` tags that define a set of data markers or series on a graph.
- `dvt:shapeAttributesSet` tag: Wraps `dvt:shapeAttributes` tags that specified 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 24–2 shows the sequence of the tags when you create a set of alert tags to define two alert points for an area graph.

Example 24–2 Sample Code 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>
```

24.3 Understanding Data Requirements for Graphs

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.

24.3.1 Area Graphs Data Requirements

An *area graph* is one in which data is represented as a filled-in area. The following kinds of area graphs are available:

- Absolute: Each area marker connects a series of two or more data values. This kind 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.

Data guidelines for area graphs are:

- Area graphs require at least two groups of data. 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.

- Area graphs require one or more series of data. 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.

24.3.2 Bar Graph Data Requirements

A *bar graph* is one in which data is represented as a series of bars. The following kinds of bar graphs are available:

- 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.

Data guidelines for bar graphs are:

- Percentage bar graphs cannot have negative numbers.

- Dual-Y graphs require two sets of data.

24.3.3 Bubble Graph Data Requirements

A *bubble graph* is one in which data is represented by the location and size of round data markers (bubbles). Each data marker in a bubble graph represents three group values:

- The first data value is the X value. It determines the marker's location along the x-axis.
- The second data value is the Y value. It determines the marker's location along the y-axis.
- The third data value is the z value. It determines the size of the marker.

The following kinds of bubble graphs are available: Bubble graph with a single y-axis and bubble graph with a dual y-axis.

Data guidelines for a bubble graph are:

- Bubble graphs require at least three data values for a data 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 more at the overall pattern of the data markers.

24.3.4 Combination Graph Data Requirements

A *combination graph* uses different types of data markers to display different sets of data. The data markers used are bar, area, and line.

Data guidelines for combination graphs are:

- 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.

24.3.5 Funnel Graph Data Requirements

A *funnel graph* is a visual representation of data related to steps in a process. As the value for a given step (or slice) of the funnel approaches the quota for that slice, the slice fills. A funnel renders a three-dimensional chart that represents target and actual values, and levels by color. A funnel graph displays data where the target is considered to be 100%. Therefore, if the actual value is 50 and target is 200, then 25% of the slice will be filled.

Data guidelines for funnel graphs are:

- Funnel graphs require two series (or sets of data). 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.

- Funnel graphs require at least one group of data to be used as a stage.

24.3.6 Line Graph Data Requirements

A *line graph* represents data as a line, as a series of data points, or as data points that are connected by a line. The following kinds of line graphs are available:

- 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.

Data guidelines for line graphs are:

- Line graphs require at least two groups of data 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.

24.3.7 Pareto Graph Data Requirements

Pareto graphs are specifically designed for identifying sources of defects. In a Pareto graph, a series of bars identifies different sources of defects. These bars are arranged by value, from the greatest number to the lowest number. A line shows the percentage of the cumulative values of the bars to the total values of all the bars in the graph. The line always ends at 100%.

Pareto graphs are always dual-Y graphs. The y1-axis corresponds to values that the bars represent. The y2-axis corresponds to the cumulative percentage values.

Data guidelines for Pareto graphs are:

- Pareto graphs require at least two groups of data.
- 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.

24.3.8 Pie Graph Data Requirements

A *pie graph* represents data as sections of one or more circles, making the circles look like sliced pies. The following varieties of pie graphs are available:

- Pie: The center of each circle is full. Pie graphs can consist of a single pie or multiple pies.

- Ring: The center of each circle has a hole in which the total pie value is displayed. Ring graphs can consist of a single ring or multiple rings.

The data structure of a pie graph 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.

Data guidelines for pie graphs are:

- Pie graphs cannot have negative numbers.
- Multiple pie graphs require at least two groups of data.

24.3.9 Polar Graph Data Requirements

A *polar graph* is a circular scatter graph. In a polar graph, as in a scatter graph, data is represented by the location of data markers. In a polar graph, the plot area, where the markers appear, is circular. For information about scatter graphs, see Section 24.3.11, "Scatter Graph Data Requirements."

Like scatter graphs, polar graphs are especially useful when you want to see general relationships among a number of data items. Use polar graphs rather than scatter graphs when the data has a directional aspect.

Each data marker in a polar graph represents two data values:

- The first data value is the X value. It determines the location of the marker along the x-axis, which is the location around the circle, clockwise.
- The second data value is the Y value. It determines the location of the marker along the y-axis, which is the distance from the center of the graph.

Data guidelines for a polar graph require at least two data values for each marker.

24.3.10 Radar Graph Data Requirements

A *radar graph* is a polygonal line graph similar to how a polar graph is a circular scatter graph. Use radar graphs to show patterns that occur in cycles, such as monthly sales for the last three years.

The data structure of a radar graph 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.

Data guidelines for radar graphs require at least three groups of data.

24.3.11 Scatter Graph Data Requirements

A *scatter graph* represents data by the location of data markers. Scatter graphs are especially useful when you want to see general relationships among a number of data points. For example, you can use a scatter graph to examine the relationships between Sales and Profit values for specific products.

Scatter graphs have either a single y-axis or a dual y-axis. Each data marker in a scatter graph represents two values:

- The first data value is the X value. It determines the marker's location along the x-axis.
- The second data value is the Y value. It determines the marker's location along the y-axis.

Data guidelines for scatter graphs are:

- Scatter graphs require two data values for each marker.
- For more than one group of data, the data must be in multiples of two.

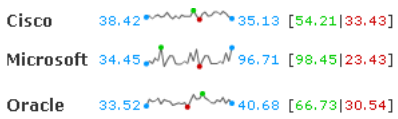
24.3.12 Sparkchart Data Requirements

Sparkcharts are used for displaying trends or variations in a single series of data values. They are condensed, simple visualizations designed to be stamped in a table or used inline with text. Since sparkcharts contain no labels, the adjacent columns of a table or surrounding text provide context for sparkchart content.

Sparkcharts do not accept tabular data or `graphDataModel`. Data guidelines for sparkcharts are:

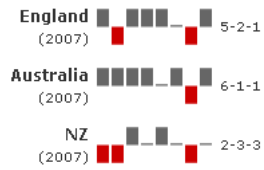
- Line, bar, and area sparkcharts require a single series of data values. Figure 24–4 shows an example of a line sparkchart in a table column.

Figure 24–4 Line Sparkchart



- Floating bar sparkcharts require two series of data values, one for the float offset, and one for the bar value. Figure 24–5 shows an example of a floating bar sparkchart in a table column.

Figure 24–5 Floating Bar Sparkchart



24.3.13 Stock Graph Data Requirements

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.

24.3.13.1 Stock Graphs: High-Low-Close

Data requirements for a high-low-close stock graph are:

- 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.
- 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 except 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.

24.3.13.2 Stock Graphs: High-Low-Close with Volume

Data requirements for a high-low-close stock graph with volume are:

- 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.

24.3.13.3 Stock Graphs: Open-High-Low-Close

Data requirements for an open-high-low-close stock graph are:

- 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.

24.3.13.4 Stock Graphs: Open-High-Low-Close with Volume

Data requirements for an open-high-low-close stock graph with volume are:

- 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.

24.3.13.5 Candle Stock Graphs: Open-Close

Data requirements for an open-close candle stock graph are:

- 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.

24.3.13.6 Candle Stock Graphs: Open-Close with Volume

Data requirements for an open-close candle stock graph with volume are:

- 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.

24.3.13.7 Candle Stock Graphs: Open-High-Low-Close

Data requirements for an open-high-low-close candle stock graph are:

- 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.

24.3.13.8 Candle Stock Graphs: Open-High-Low-Close with Volume

Data requirements for an open-high-low-close candle stock graph with volume are:

- 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.

24.4 Creating a Graph

You can use any of the following data sources to create a graph component:

- ADF Data Controls: You declaratively create a databound graph by dragging and dropping a data collection from the ADF Data Controls panel. You can create a graph using a data collection that provides row set data as described in the "Creating Databound ADF Graphs" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Hierarchical data: You can create a graph from a data control that provides hierarchical data. However, the current release does not include an implementation of a hierarchical data control that is supported by graph.
- Tabular data: You can provide CSV (comma-separated value) data to a graph through the `tabularData` attribute as shown in Section 24.4.1, "How to Create a Graph Using Tabular Data.".

24.4.1 How to Create a Graph Using Tabular Data

The process of creating a graph from tabular data includes the following steps:

- Storing tabular data in a method in the graph's managed bean.
- Creating a graph that uses the tabular data stored in the managed bean.

24.4.1.1 Storing Tabular Data for a Graph in a Managed Bean

The `tabularData` attribute of a `dvt:graph` component lets you specify a list of data that the graph uses to create a grid and populate itself. To construct this list, you require an understanding of series and groups of data in a graph as well as knowledge of the structure of the list.

24.4.1.1.1 Series and Groups of Data A graph displays series and groups of data. Series and groups are analogous to the rows and columns of a grid. Usually 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. Usually 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. 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.

24.4.1.1.2 Structure of the List of Tabular Data 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`.

24.4.1.1.3 Example of a List of Data Figure 24–6 has three columns: 2006, 2007, and 2008. This graph also has two row: Shoes and Boots. This data produces a graph that compares annual sales for boots and shoes over a three-year period.

Figure 24–6 Comparison of Annual Sales

	2006	2007	2008	
Shoes	120000	122000	175000	
Boots	90000	110000	150000	

Example 24–3 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 24–3 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, 122000, 175000},
        {90000, 110000, 150000}
        };
    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;
}
```

24.4.1.2 Creating a Graph Using Tabular Data

Use the `tabularData` attribute of a graph tag to reference data that is stored in a method in a managed bean.

To create a graph that uses data from a managed bean:

1. In the Structure window, right-click the **graph** node and choose **Go to Properties**.
2. In the **Data** attributes category of the Property Inspector, click the **TabularData** attribute dropdown menu and choose **Expression Builder**.
3. From the ensuing dialog, use the search box to locate the managed bean.
4. Expand the managed bean node and select the method that contains the list of tabular data.
5. Click **OK**.

In the Expression Builder, the `tabularData` attribute is set to reference the method that you selected in the managed bean. For example, for a managed bean named named `sampleGraph` and a method named `getTabularData`, the `tabularData` attribute has the following setting: `#(sampleGraph.tabularData)`.

24.4.2 What Happens When You Create a Graph Using Tabular Data

When you create a graph that is powered by data obtained from a list referenced the `tabularData` attribute a vertical clustered bar graph is created by default. You have the option of changing the settings of the `graphType` attribute to any of the more than 50 graphs that are available as long as the tabular data meets the data requirements for that graph type. You can also change the settings of the many additional attributes on the `dvt:graph` tag.

Customize the graph by dragging any of the graph child tags to the `dvt:graph` node in the Structure window and providing settings for the attributes that you want to specify.

24.4.3 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. By default, graphs display in Flash, but you can configure your application to use a specific image format by setting the following parameters:

- `oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT`

  To use the HTML5 image format, add this parameter to the `web.xml` file and set it to `HTML5`. For more information, see Section A.2.3.23, "Graph and Gauge Image Format."

- `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 isn't 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 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

24.5 Changing the Graph Type

When you insert a graph using the Data Controls panel or the Component Palette, the Component Gallery displays available graph categories, types, and quick-start layout

options from which to choose. Selecting a graph type sets the `subType` attribute for that graph. You can change the type for all graphs except the funnel and radar graphs.

To change the type of a graph:

1. In the Structure window, right-click the **graph** node and choose **Go to Properties**.
2. In the **Common** attributes category of the Property Inspector, for the **SubType** attribute, 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.

24.6 Customizing the Appearance of Graphs

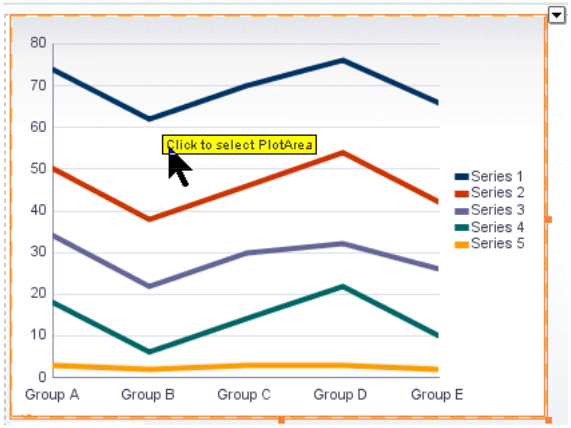
Most graph types have common features that are available for customization. The following types of customization are supported by most graph types:

- Changing the color, style, and display of graph data values. For information, see Section 24.6.1, "Changing the Color, Style, and Display of Graph Data Values."
- Formatting categorical and numerical data values in graphs. For information, see Section 24.6.2, "Formatting Data Values in Graphs."
- Formatting text in graphs. For information, see Section 24.6.3, "Formatting Text in Graphs."

> **Note:** In order to avoid invalid color values, JDeveloper provides a color selection dialog when you specify color-related attributes in graph elements.

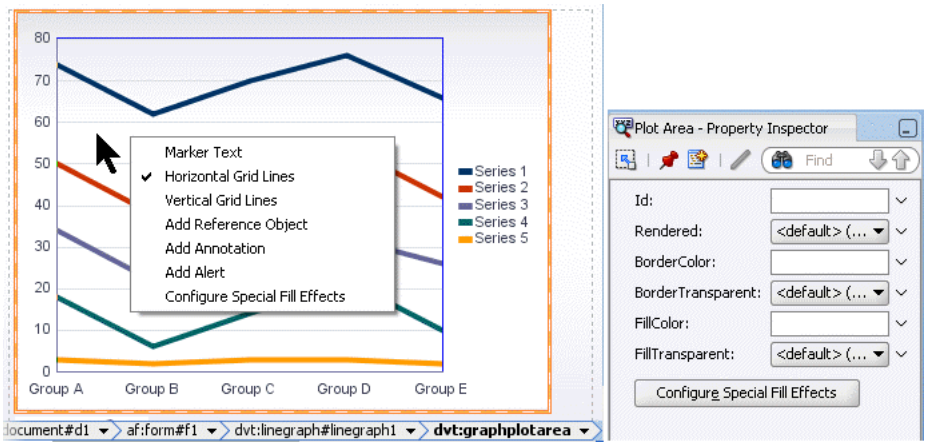
When you edit graph components in the visual editor, specialized context menus and Property Inspector buttons are available to support the customization of graph features. Popups in the editor provide confirmation of selection of the graph feature to be customized. For example, Figure 24–7 shows the popup displayed in the plot area of a line graph.

Figure 24–7 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 24–8 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 Property Inspector buttons to configure special fill effects in the plot area. The selection of the graph tags is synchronized in the visual editor, Structure window, Property Inspector, and source editor.

Figure 24–8 Line Graph Plot Area Context Menu



For additional information about configuring line graphs, see. Section 24.7.2, "Changing the Appearance of Lines in Graphs." For additional information about configuring special fill effects, see Section 24.8.2, "Using Gradient Special Effects in Graphs."

24.6.1 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 24.7.2, "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 24.7.4, "Customizing Scatter Graph Series Markers."

You can also customize the colors of each series in a graph by adding gradient special effects. For more information, see Section 24.8.2, "Using Gradient Special Effects in Graphs."

24.6.1.1 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.

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 Property Inspector 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, right-click the **seriesSet** node and choose **Insert inside dvt:seriesSet** > **Series**.

   The first `dvt:series` tag represents the first series item that appears in the Create Graph Binding dialog.
4. Use the Property Inspector to specify colors and other characteristics as needed for the `dvt:series` tag.
5. Repeat Step 3 and Step 4 for each series item.

24.6.1.2 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.

To enable hiding and show series items:

1. In the Structure window, right-click the **graph** node and choose **Go to Properties**.
2. In the Property Inspector, 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.

24.6.2 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 6, "Validating and Converting Input."

24.6.2.1 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 24–4 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 24–4 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 hire date and bonus categorical data values defined in the page definition file in Example 24–4.

To format categorical data values defined in a page definition file:

1. In the Structure window, right-click the bar graph tag and choose **Insert inside dvt:barGraph > ADF Data Visualizations > Attribute Format**.
2. In the Property Inspector, for the **Name** attribute, enter `Hiredate` as the name of the **af1** category attribute.
3. In the Structure window, right-click the attribute format tag you named and choose **Insert inside dvt:attributeFormat > Convert Date Time**.

4. In the Property Inspector, for the **Pattern** attribute, enter the formatting pattern for the date/time string conforming to `java.text.SimpleDateFormat`. For the **TimeZone** attribute, enter the timezone to interpret any time information in the data string.
5. Repeat Steps 1-4 for the bonus category attribute, setting `Bonus` as the name of the **af2** category attribute, adding an `af:convertNumber` converter, and formatting the attribute for currency.

Example 24–5 shows the XML code that is generated if you format the categorical data values in a bar graph.

Example 24–5 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.

24.6.2.2 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. Specify the `textType` attribute of the `dvt:sliceLabel` tag to display the value represented in the slice, and format the number accordingly.

To format numbers in the slice label of a pie graph:

1. In the Structure window, right-click the child **dvt:sliceLabel** tag of the pie graph tag and choose **Go to Properties**.
2. In the Property Inspector, choose **LD_VALUE** from the **TextType** attribute dropdown list to specify that the pie slice in the graph represents a value.
3. In the Property Inspector, click **Configure Slice Label** and choose **Number Format** from the dropdown list.
4. In the Property Inspector, for the **af:convertNumber** tag, specify the values as currency, using a dollar sign as the currency symbol.

Example 24–6 shows the XML code that is generated if you format the numerical data values in the slice label of a pie graph to appear as currency, and use the dollar sign symbol.

Example 24–6 Formatting Numerical Data Values 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:y1Format`.

To format numerical values in the marker text associated with the y1-axis of a graph:

1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Marker Text**.
2. In the Property Inspector, 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 Property Inspector, click **Configure Marker** and choose **Y1 Format**.
4. In the Property Inspector, optionally enter values as needed for the **dvt:y1Format** attributes.
5. In the Property Inspector, 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 24–7 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 the text appears above the markers. For example, in a bar graph, the text will appear above the bars.

Example 24–7 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.

24.6.2.3 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`.

24.6.3 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 24.6.5.2, "How to Specify Titles and Footnotes in a Graph,".

24.6.3.1 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.

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:
}
```

2. Register the custom skin with the application in the `trinidad-skins.xml` file. For example, `mySkin.css` extends the default `blafplus-rich.desktop` style sheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
  <skin>
    <id>mySkinExtends.desktop</id>
    <family>mySkinExtends</family>
```

```
    <extends>blafplus-rich.desktop</extends>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <style-sheet-name>skins/mySkin.css</style-sheet-name>
  </skin>
</skins>
```

3. Configure the application to use the custom skin in the `trinidad-config.xml` file. For example:

```
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>mySkin</skin-family>
</trinidad-config>
```

4. Package the custom skin into a jar file to deploy with the application. The `trinidad-skins.xml` file that defines the skin and that references the CSS file must be within the `META-INF` directory.

For detailed information about applying a custom skin to applications, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

24.6.4 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.

24.6.4.1 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.

To specify the size of a graph at its initial display:

1. In the Structure window, right-click the graph node and choose **Go to Properties**.
2. In the Style attributes category of the Property Inspector, enter a value for the `inlineStyle` attribute of the graph tag. For example:

```
inlineStyle="width:200px;height:200px"
```

24.6.4.2 How to Provide for Dynamic Resizing of a Graph

You must enter values in each of two attributes of the `dvt:<type>Graph` 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.

To allow dynamic resizing of a graph:

1. In the Structure window, right-click the graph node and choose **Go to Properties**.
2. In the Behavior attributes category of the Property Inspector for the `DynamicResize` attribute, select the value **DYNAMIC_SIZE**.
3. In the Style attributes category of the Property Inspector 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 its container's width and has a height of 200 pixels, use the following setting for the `inlineStyle` attribute: `"width:100%;height:200px;".`

24.6.4.3 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:<type>Graph` tag. You can also specify a custom style sheet for use with a graph.

To select a specific style sheet for a graph:

1. If you want to use one of the standard style sheets provided with the graph, do the following:
   a. In the Structure window, right-click the graph node and choose **Go to Properties**.
   b. In the Appearance attributes category, select the desired style sheet from the `style` attribute dropdown list.
2. If you want to use a custom style sheet, then set the following attributes in the Style attributes category of the Property Inspector:
   a. For the `StyleClass` attribute, select Edit from the Property menu choices, and select the CSS style class to use for this gauge.
   b. In 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 its container's width and has a height of 200 pixels, use the following setting for the `inlineStyle` attribute: `"width:100%;height:200px;"`

24.6.5 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.

24.6.5.1 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.

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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Background**.

b. Use the Property Inspector to enter colors in the attributes that you want to customize in the `dvt:background` tag.

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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Plot Area**.

   b. Use the Property Inspector to enter colors in the attributes that you want to customize in the `dvt:graphPlotArea` tag.

3. If you want to customize the plot area of a pie graph, do the following:

   a. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Pie Frame**.

   b. Use the Property Inspector to enter colors in the attributes that you want to customize in the `dvt:graphPieFrame` tag.

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 24.8.2, "Using Gradient Special Effects in Graphs."

24.6.5.2 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.

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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Graph Title**.

   b. Use the Property Inspector 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 dvt:graphTitle** > **Font**.

   d. Use the Property Inspector 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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Subtitle**.

   b. Use the Property Inspector 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 dvt:graphSubtitle** > **Font**.

d. Use the Property Inspector 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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Footnote**.

   b. Use the Property Inspector 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 dvt:graphFootnote** > **Font**.

   d. Use the Property Inspector to specify values for the attributes of the `dvt:graphFont` tag.

24.6.6 Customizing 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.

- 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.

24.6.6.1 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.

To specify the title and appearance of an x1-axis:

1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **x1 Title**.
2. In the Property Inspector, 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:x1Title** node and choose **Insert inside dvt:x1Title** > **Font**.
   b. In the Property Inspector, enter the desired values for the characteristics of the font.
4. Optionally, in the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **x1 Axis**.
5. In the Property Inspector, enter the desired values for the attributes of this tag.

The procedure for controlling the title and appearance of any graph axis is similar to the procedure for the x-axis. However, insert the title and axis tags related to the specific axis that you want to customize.

24.6.6.2 How to Specify 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.

24.6.6.3 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.

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` element in the Structure window, then in the Property Inspector click the **Configure o1Axis** button and choose **Value Labels**.

2. In the Property Inspector enter values as needed for the following properties:

   - **TextRotation**: Use to specify the degree of text rotation to improve readability of the tick labels.

     Note: Use rotation angles that are multiples of 90 degrees to achieve best results. For Flash image types, embedded fonts are required to support rotated text in non-90 degree angles, and embedded fonts are not available for all locales.

   - **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` to set the number of tick labels to display between tick labels and `tickLabelSkipFirst` to set the index of the first tick label to be skipped.

3. Optionally, in the Property Inspector, 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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **X1 Major Tick**.
2. In the Property Inspector, 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 Property Inspector, 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 dvt:<type>Graph** > **ADF Data Visualization** > **X1 Minor Tick**.
   b. In the Property Inspector, 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.

24.6.6.4 How to Format 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`.

To format numbers on these axes, insert child tags for the appropriate axis as shown in Section 24.6.2, "Formatting Data Values in Graphs."

24.6.6.5 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.

To specify the starting value on a y1-axis:

1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **y1 Axis**.
2. In the Property Inspector, 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 insert the `dvt:y2Axis` tag as a child of the graph.

24.6.7 Customizing 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

To customize the legend area, legend text, and title:

1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Legend Area**.

2. Use the Property Inspector 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**: 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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Legend Text**.
   b. Use the Property Inspector to enter values for the attributes of this tag.
   c. Right-click the **dvt:legendText** node and choose **Insert inside dvt:legendText** > **Font**.
   d. Use the Property Inspector 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 graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization**, then **Legend Title**.
   b. Use the Property Inspector to enter values for the attributes of this tag.
   c. Right-click the **dvt:legendTitle** node and choose **Insert inside dvt:legendTitle** > **Font**.
   d. Use the Property Inspector to specify values for the attributes of the font tag.

24.6.8 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 displays 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.

- **SeriesTooltipLabel**: 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 24–1.

Table 24–1 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.

Table 24–1 (Cont.) markerTooltipTemplate String Tokens

Token	Description
CUM_PERCENT	The cumulative percentage value for a stacked percent graph or Pareto graph.

You can use the graph's `customToolTipCallback` attribute to specify tooltips that vary on an object by object basis. For example:

```
<dvt:graph id="g2" customToolTipCallback="#{customToolTipCallback.callback}"
```

24.7 Customizing the Appearance of Specific Graph Types

The graph components support more than 50 graph types. Some of the graph attributes and several child tags relate only to specific graph types.

24.7.1 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.

24.7.1.1 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 as a child to the `dvt:sliceLabel` tag: `dvt:graphFont` and `af:convertNumber`.

24.7.1.2 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 24.6.1.1, "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. To separate one slice in a pie graph, in the Property Inspector, for the `series` tag that represents the pie slice that you want to separate from the pie, set the **PieSliceExplode** attribute between 0 to 100, where 100 is the maximum exploded distance available.

2. To animate the slices in a pie graph, in the Property Inspector, 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 use 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.

24.7.2 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.

24.7.2.1 How to Display 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 Property Inspector, 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`.

24.7.2.2 How to Change 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, and combination graphs. Identifies a default marker shape for all the series in the series set.
  - `defaultMarkerType`: Used only for combination 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 24.6.1.1, "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.

24.7.3 Customizing 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.

To customize a Pareto graph:

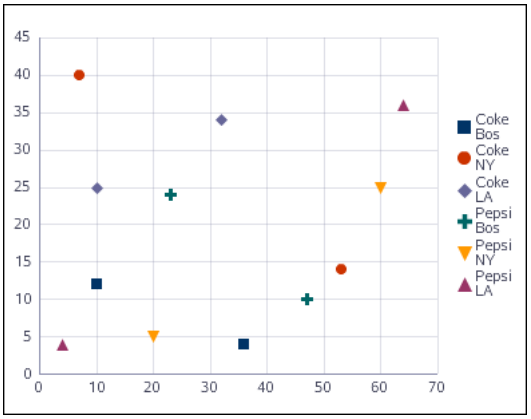
1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Pareto Line**.
2. In the Property Inspector, specify values for the attributes of this tag.
3. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph > ADF Data Visualization** > **Pareto Marker**.
4. In the Property Inspector, select a value for the `markerShape` attribute.

24.7.4 Customizing Scatter Graph Series Markers

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.

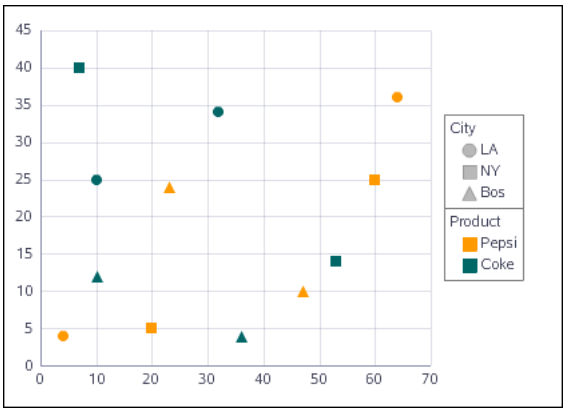
For example, Figure 24–9 shows a scatter graph that uses City and Product attributes collectively to determine the series represented by the data marker's shape and color.

Figure 24–9 Scatter Graph with Series Marker



The row header attributes can be used to override the default series specification. Figure 24–10 shows a scatter graph that displays the data values for the City attribute mapped to shapes and the Product attribute mapped to colors.

Figure 24–10 Scatter Graph with Series Item Markers



Use the following attributes to customize the scatter graph series markers:

- `markerShape` - 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.
- `markerColor` - 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.

For example, specify City and Product as separate series item markers using this code:

```
<dvt:scatterGraph markerColor="Product" markerShape="City"
value="#{bindings.View1.graphModel}"/>
```

24.8 Adding Specialized Features to Graphs

There are graph customization features that include the ability to define series-related reference lines and axis-related reference areas, the option of adding gradient special effects to several parts of a graph, the option of setting some parts of a graph to transparent colors, and the use of alerts and annotations in graphs. These special features also let you use the interactive capabilities of the graph such as providing

context menus, reacting to changes in the zoom or scroll levels, and reacting to user clicks on the graph's data markers.

24.8.1 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.

24.8.1.1 How to Create Reference Lines or Areas During Design

Both reference lines and reference areas are created by the use of the following tags:

- `dvt:referenceObjectSet`: Wraps all the reference object tags for reference lines or reference areas for this graph.
- `dvt:referenceObject`: Identifies whether the tag represents a reference line or a reference area and specifies characteristics for the tag.

To add reference lines or areas to a graph during design:

1. In the Structure window, right-click the graph node and choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Reference Object Set**.
2. If you are defining reference areas related to specific axes, then specify a value for 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 the time.

   Optionally, you can apply a gradient special effect to the reference area. For more information see Section 24.8.2.1, "How to Add Gradient Special Effects to a Graph."
3. In the Structure window, right-click the `dvt:referenceObjectSet` node and choose **Insert inside dvt:referenceObjectSet** > **Reference Object**.
4. In the Property Inspector, do the following:
   a. In the Common attributes category, specify values for the `index` attribute of the reference object, the `type` attribute of the reference object (`RO_LINE` or `RO_AREA`), 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. This includes specifying the 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 that represent the reference area on the specified axis.

5. In the Structure window, right-click the graph node and choose **Go To Properties**.
6. In the Property Inspector, select the **Appearance** attributes category and do the following to control the display of reference lines and reference areas:

   a. If you have defined reference lines (which must be related to a series), then expand the **dvt:series** node and specify a value for the behavior of the `displaySeries` attribute.

      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.

   b. If you have defined reference areas (which must be related to specific axes), then associate the reference object with the appropriate axis or axes.

24.8.1.2 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 24–8 shows the code for three reference areas associated with the y1-axis, one reference area associated with the y2-axis, and four reference lines associated with different series.

Example 24–8 XML Code for Reference Lines and Areas in a Graph

```
<dvt:barGraph value ="#{sampleGraph.graphDataModel}" graphType="BAR_VERT_CLUST2Y"
     imageFormat="FLASH">
  <dvt:referenceObjectSet
     displayX1="RO_DISPLAY_AUTOMATIC"
     displayY2="RO_DISPLAY_AUTOMATIC"
    <dvt:referenceObject index="1" type="RO_AREA" association="Y1AXIS"
     location="RO_BACK" color="#55FF0000" lowValue="0" highValue="4000"
     displayedInLegend="true" text="Low"/>
    <dvt:referenceObject index="2" type="RO_AREA" association="Y1AXIS"
      location="RO_BACK" color="#55FFFF00"  lowValue="4000" highValue="10000"
      displayedInLegend="true" text="Medium"/>
    <dvt:referenceObject index="3" type="RO_AREA" association="Y1AXIS"
      location="RO_BACK" color="#5500FF00"  lowValue="10000" highValue="18000"
      displayedInLegend="true" text="High"/>
    <dvt:referenceObject index="4" type="RO_AREA" association="Y2AXIS"
      location="RO_FRONT" color="#550000FF" lowValue="300" highValue="700"/>
    <dvt:referenceObject index="5" type="RO_LINE" association="SERIES" series="0"
      location="RO_FRONT" color="#ffff66" lineValue="5000" lineWidth="3"
      lineStyle="LS_SOLID"/>
    <dvt:referenceObject index="6" type="RO_LINE" association="SERIES" series="0"
      location="RO_FRONT" color="#ffff66" lineValue="16730" lineWidth="3"
      lineStyle="LS_SOLID"/>
    <dvt:referenceObject index="7" type="RO_LINE" association="SERIES" series="1"
      location="RO_BACK" color="#99cc66" lineValue="500" lineWidth="3"
      lineStyle="LS_SOLID"/>
    <dvt:referenceObject index="8" type="RO_LINE" association="SERIES" series="1"
      location="RO_BACK" color="#99cc66" lineValue="1711" lineWidth="3"
```

```
      lineStyle="LS_SOLID"/>
  </dvt:referenceObjectSet>
</dvt:barGraph>
```

24.8.1.3 How to Create Reference Lines or Areas Dynamically

If you want to create reference objects dynamically at runtime, then you use only the `dvt: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.

To create reference lines or areas dynamically:

1. Write a method that creates a map of the child component reference objects that you want to create during runtime. Example 24–9 shows sample code for creating this method.
2. In the Structure window, right-click the graph node, then choose **Insert inside dvt:<type>Graph** > **ADF Data Visualization** > **Reference Object Set**.
3. In the Property Inspector, 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`, the attribute should be set to the following value:
   `referenceObjectMap="#{sampleGraph.referenceObjectMapList}"`

Example 24–9 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;
    }
```

24.8.2 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.

24.8.2.1 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. For example, if you want to add a gradient to the background of a graph, then you would create one `dvt:specialEffects` tag that is a child of the `dvt:background` 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.

To add a gradient special effect to the background of a graph:

1. In the Structure window, right-click the `dvt:background` node that is a child of the graph node, then choose **Insert inside dvt:background,** then **Special Effects**.
2. Use the Property Inspector to enter values for the attributes of the `dvt:specialEffects` tag:
   a. For `fillType` attribute, choose **FT_GRADIENT**.

      For `gradientDirection` attribute, select the direction of change that you want to use for the gradient fill.

   b. For `numStops` attribute, enter the number of stops to use for the gradient.
3. Optionally, in the Structure window, right-click the `dvt:specialEffects` node and choose **Insert within dvt:specialEffects > dvt:gradientStopStyle** if you want to control the color and rate of change for each gradient stop.
4. Use the Property Inspector to enter values for the attributes of the `dvt:gradientStopStyle` tag:
   a. For the `stopIndex` attribute, enter a zero-based integer as an index within the `dvt:gradientStopStyle` tags that are included within the `specialEffects` tag.

   b. For the `gradientStopColor` attribute, enter the color that you want to use at this specific point along the gradient.

   c. For the `gradientStopPosition` attribute, 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.
5. Repeat Step 3 and Step 4 for each gradient stop that you want to specify.

24.8.2.2 What Happens When You Add a Gradient Special Effect to a Graph

Example 24–10 shows the XML code that is generated when you add a gradient fill to the background of a graph and specify two stops.

Example 24–10 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>
```

24.8.3 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. The following list identifies the parts of the graph that support transparency:

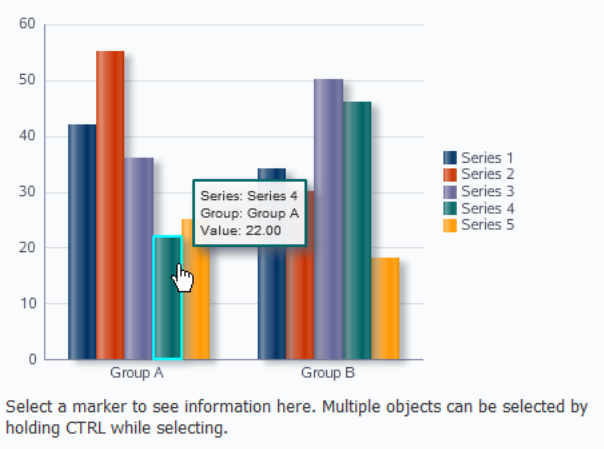
- Graph background: Use the `dvt:background` tag. By default the `fillTransparent` attribute is set to `true`.
- Graph legend area: Use the `dvt:legendArea` tag.
- Graph pie frame: Use the `dvt:graphPieFrame` tag.
- Graph plot area: Use the `dvt:graphPlotArea` tag.

24.8.4 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, such as bubbles in a bubble graph or shapes in a scatter graph.

Figure 24–11 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.

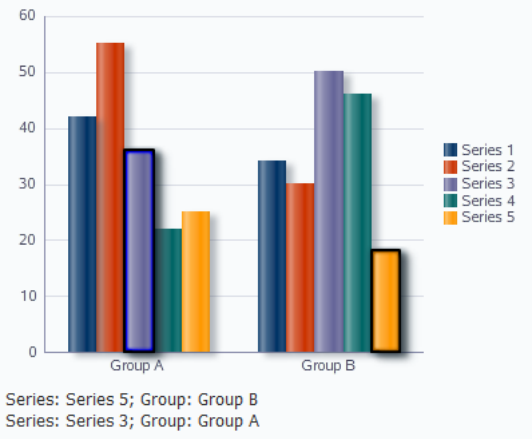
Figure 24–11 Bar Graph With Selection Support Enabled



Graphs can be enabled for single and multiple selection support. Enabling selection is required for context menus and for responding programmatically to user clicks on the data markers.

For example, Figure 24–12 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 24–12 Bar Graph Displaying Multiple Selection Support



24.8.4.1 How to Add Selection Support to Graphs

To add selection support, create a listener in a managed bean that will handle the selection event 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 24–11 shows sample code to create a managed bean that returns the selection state as the formatted string displayed below the bar graph in Figure 24–12.

Example 24–11 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 24–12 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 24–12 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>
```

To Add Selection Support to Graphs:

1. 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 2.6.1, "How to Create a Managed Bean in JDeveloper."
2. Add a graph to your page. For more information, see Section 24.4, "Creating a Graph."

   To duplicate the multiple selection support example in this section, create a bar graph.
3. In the Structure window, right-click the **dvt:<type>Graph** node and choose **Go to Properties**.
4. In the Property Inspector, 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}`.
5. 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}"/>
   ```

24.8.4.2 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.

24.8.5 Adding Context Menus to Graphs

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

24.8.5.1 How to Configure Graph Context Menus

Context menus can be defined for graph components using these context menu facets:

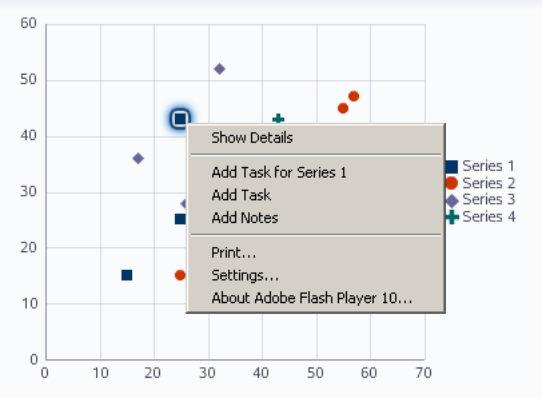
- `bodyContextMenu`: Specifies a context menu that is displayed on non selectable elements in the graph component.
- `contextMenu`: Specifies a context menu that is displayed on any selectable element in the graph component.
- `multiSelectContextMenu`: Specifies a content menu that is displayed when multiple elements are selected in the graph component.

Each facet supports a single child component. 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 24.8.5.2, "What You May Need to Know About Flash Rendering Format."

For example, Figure 24–13 shows a scatter graph context menu with custom menu items.

Figure 24–13 Scatter Graph Custom Context Menu



Example 24–13 shows a code sample for configuring a scatter graph context menu.

Example 24–13 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 24–14 shows a code sample for a managed bean to create a custom context menu. For help with managed beans, see Section 2.6, "Creating and Using Managed Beans."

Example 24–14 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 24–15. Store the code for this class in an additional managed bean.

Example 24–15 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;
  }
}
```

24.8.5.2 What You May Need to Know About Flash Rendering Format

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.

24.8.6 How to React to Changes in the Zoom and Scroll Levels

You can provide custom code that will be executed when the zoom and scroll levels change on a graph. In a managed bean you store methods that takes as input a ZoomEvent or ScrollEvent. With these events, users can determine which axis is zoomed, as well as the current extent of the zoomed axes.

To provide custom behavior in response to zooming and scrolling in a graph:

1. In a managed bean, write a custom method that performs the desired behavior when a zoom or scroll event is triggered. Example 24–16 shows sample code for creating this method.
2. In the Structure window, right-click the graph node and choose **Go to Properties**.
3. Select the **Behavior** attributes category and expand the **Advanced** node and do one or both of the following:
   - In the zoomlListener 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.zoom)".
   - 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.scroll)".

Example 24–16 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));
```

24.8.7 How to Provide Marker and Legend Dimming

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. Markers include lines, bars, areas, scatter markers, bubbles, and pie slices. 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, then 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.

To dim all the data markers in a series:

1. In the Structure window, right-click the graph node and choose **Go to Properties**.
2. In the Appearance attributes category, in the `SeriesRolloverBehavior` field, use the dropdown list to select **RB_DIM**.

24.8.8 Providing an Interactive Time Axis for Graphs

You can define relative ranges and explicit ranges for the display of time data.

24.8.8.1 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.

To specify a relative range of time data for display:

1. In the Structure window, right click the graph node and choose **Go to Properties**.
2. In the Appearance attributes category, expand the **dvt:timeAxis** node and specify values for the following attributes:
   a. In the `timeRangeMode` attribute, 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. In the `timeRelativeRange` attribute, specify the relative range in milliseconds.

24.8.8.2 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.

To specify an explicit range of time data for display:

1. In the Structure window, right click the graph node and choose **Go to Properties**.
2. In the Appearance attributes category, specify the values for the following attributes:

a. In the `timeRangeMode` attribute, specify the value **TRM_EXPLICIT**.

b. In the `timeRangeStart` attribute, enter the initial date for the time range.

c. In the `timeRangeEnd` attribute, enter the ending date for the time range.

24.8.9 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. An unlimited number of alerts can be defined for a graph using `dvt:alert` tags. The alerts are wrapped in a `dvt:alertSet` tag, that is a child of graph tag. Example 24–17 shows a set of alerts for an area graph.

Example 24–17 Sample Code for Set of Graph Alerts

```
<dvt:areaGraph>
  <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:areaGraph>
```

Annotations are associated with a data value on a graph to provide information when the cursor moves over the data value. 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 24–18 shows a set of annotations for an area graph.

Example 24–18 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.

24.9 Animating Graphs

Graph components dvt:areaGraph, dvt:bubbleGraph, dvt:barGraph, dvt:lineGraph, dvt:comboGraph, dvt:pieGraph, and dvt:scatterGraph 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:

- alphaFade
- conveyorFromLeft
- conveyorFromRight
- cubeToLeft
- cubeToRight
- flipLeft
- flipRight
- slideToLeft
- slideToRight
- transitionToLeft
- transitionToRight
- zoom

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.

24.9.1 How to Configure Graph Components to Display Active Data

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 the "Using the Active Data Service" chapter in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Configure databound graphs to display active data by setting a value on the binding element in the corresponding page definition file.

To configure a graph to display active data:

1. In the Structure window, select the graph node.
2. In the Property Inspector, enter a unique value in the ID field.

   If you do not select an identifier, one will be entered for you.
3. Open the page's associated page definition file.
4. In the Structure window for the page definition file, select the node that represents the attribute binding for the component. In the Property Inspector, select **Push** for the `ChangeEventPolicy` attribute.

24.9.2 How to Specify Animation Effects for Graphs

In the Property Inspector for the graph you wish to animate, set the following attributes:

- `animationOnDisplay`: Optional. Use with or without ADS 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.
  - `alphaFade`
  - `conveyorFromLeft` or `conveyorFromRight`
  - `cubeToLeft` or `cubeToRight`
  - `flipLeft` or `flipRight`
  - `slideToLeft` or `slideToRight`
  - `transitionToLeft` or `transitionToRight`
  - `zoom`
- `animationOnDataChange`: Use to specify the type of data change animation to apply. Valid values are:
  - `none`: Apply no data change animation effects.
  - `activeData` (default): Apply Active Data Service (ADS) data change animation events.

  - 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.

25

Using ADF Gauge Components

This chapter describes how to use a databound ADF gauge component to display data, and provides the options for gauge customization.

This chapter includes the following sections:

- Section 25.1, "Introduction to the Gauge Component"
- Section 25.2, "Understanding Data Requirements for Gauges"
- Section 25.3, "Creating a Gauge"
- Section 25.4, "Customizing Gauge Type, Layout, and Appearance"
- Section 25.5, "Adding Gauge Special Effects and Animation"
- Section 25.6, "Using Custom Shapes in Gauges"

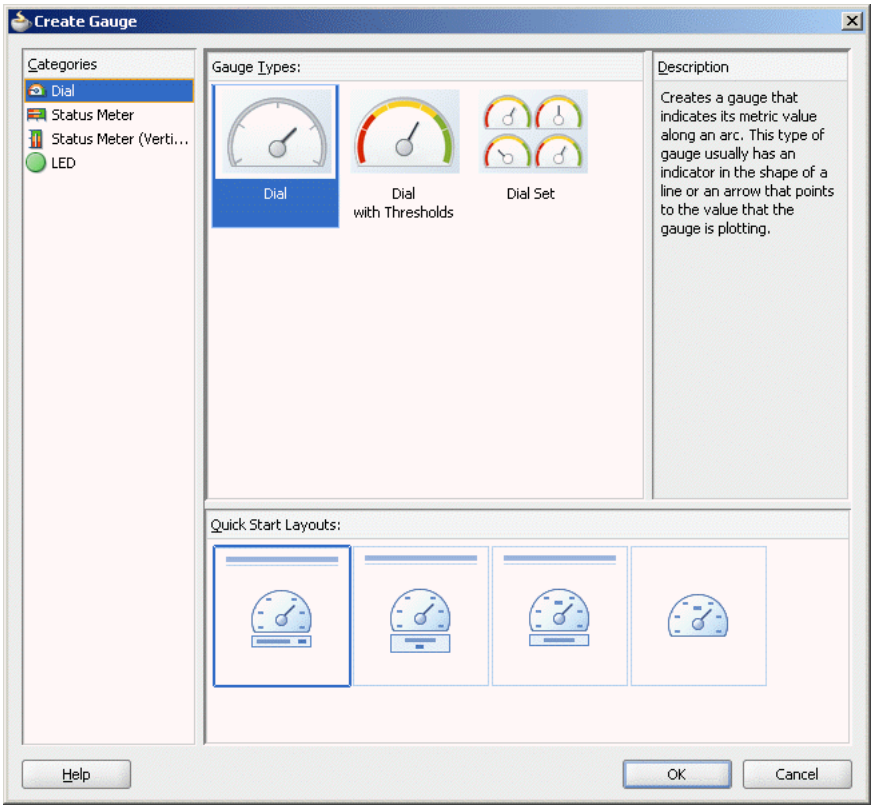
For information about the data binding of gauges, see the "Creating Databound ADF Gauges" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

25.1 Introduction to the Gauge Component

Gauges identify problems in data. A gauge usually plots one data point with an indication of whether that point falls in an acceptable or an unacceptable range. Frequently, you display multiple gauges in a single *gauge set*. The gauges in a set usually appear in a grid-like format with a configurable layout.

A Component Gallery displays available gauge categories, types, and descriptions to provide visual assistance when creating gauges and using a quick-start layout. Figure 25–1 shows the Component Gallery for gauges.

Figure 25–1 Component Gallery for Gauges



When a gauge component is inserted into a JSF page using the Component Gallery, a set of child tags that support customization of the gauge is automatically inserted. Example 25–1 shows the code inserted in the JSF page for a dial gauge with the quick-start layout selected in the Component Gallery in Figure 25–1.

Example 25–1 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>
```

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.

By default, gauges are displayed using a Flash image format. Alternatively, gauges can be displayed using HMTL5 or a Portable Network Graphics (PNG) output format. For

more information about gauge image formats, see Section 25.3.4, "What You May Need to Know About Gauge Image Formats."

HTML5, Flash, and PNG image formats for gauges support bi-directional locales. Support includes text strings containing bi-directional characters, label positions, legend display, and gauge set display.

25.1.1 Types of Gauges

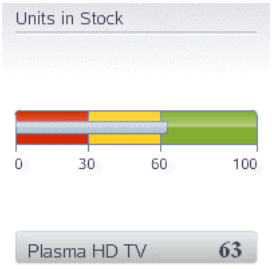
The following types of gauges are supported by the gauge component:

- Dial: Indicates its metric along a 220 degree arc. This is the default gauge type. Figure 25–2 shows a dial gauge indicating a Plasma HD TV stock level within an acceptable range.

Figure 25–2 Dial Gauge with Thresholds

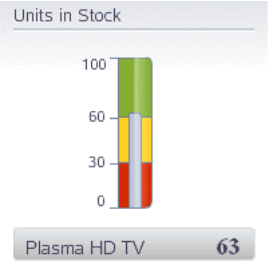
- Status Meter: Indicates the progress of a task or the level of some measurement along a rectangular bar. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Figure 25–3 shows the Plasma HD TV stock level using a status meter gauge.

Figure 25–3 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. Figure 25–4 shows the Plasma HD TV stock level using a vertical status meter gauge.

Figure 25–4 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 arrows that indicate good (up arrow), fair (left- or right-pointing arrow), or poor (down arrow). Figure 25–5 shows the Plasma HD TV stock level using a LED bulb indicator.

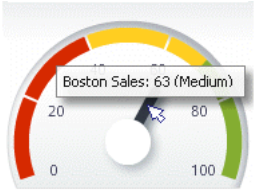
Figure 25–5 LED Bulb Gauge

Figure 25–6 shows the same stock level using a LED arrow.

Figure 25–6 LED Arrow Gauge

For dial and status meter gauges, a tooltip of contextual information automatically displays when a users moves a mouse over the plot area, indicator, or threshold region. Figure 25–7 shows the indicator tooltip for a dial gauge.

Figure 25–7 Indicator Tooltip for Dial Gauge



25.1.2 Gauge Terminology

Gauge terms identify the many aspects of a gauge and gauge set that you can customize. The gauge component includes approximately 20 child tags that provide options for this customization.

The parts of a gauge that can be customized are:

- Overall gauge customization: Each item in this group is represented by a gauge child tag:
  - Gauge Background: Controls border color and fill color for the background of a gauge.
  - Gauge Set Background: Controls border color and fill color for the background of a gauge set.
  - Gauge Frame: Refers to the frame behind the dial gauge.
  - Plot Area: Represents the area inside the gauge itself.
  - Indicator: Points to the value that is plotted in a dial gauge. It is typically in the form of a line or an arrow.
  - Indicator Bar: The inner rectangle in a status meter gauge.
  - Indicator Base: The circular base of a line or needle style indicator in a dial gauge.
  - Threshold Set: Specifies the threshold sections for the metrics of a gauge. You can create an infinite number of thresholds for a gauge.
- Data values: These include the metric (which is the actual value that the gauge is plotting), minimum value, maximum value, and threshold values. Section 25.2, "Understanding Data Requirements for Gauges" describes these values.
- Labels: The gauge supports the following elements with a separate child tag for each item:
  - Bottom Label: Refers to an optional label that appears below or inside the gauge. By default displays the label for the data row.
  - Lower Label Frame: Controls the colors for the background and border of the frame that contains the bottom label. The metric label can also appear inside the lower label frame, to the right of the bottom label.
  - Metric Label: Shows the value of the metric that the gauge is plotting in text.
  - Tick Marks: 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 labels or minor increments.
  - Tick Labels: Displays text that is displayed to identify major tick marks on a gauge.
  - Top Label: Refers to the label that appears at the top or inside of a gauge. By default, a title separator is used with a label above the gauge. By default, displays the label for the data column.
  - Upper Label Frame: Refers to the background and border of the frame that encloses the top label. You can specify border color and fill color for this frame. Turn off the default title separator when using this frame.
- Legend: The gauge supports the gauge legend area, text, and title elements with a separate child tag for each item.

- Shape Attributes Set: The gauge supports interactivity properties for its 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 25.5.3, "How to Add Interactivity to Gauges."

25.2 Understanding Data Requirements for Gauges

You can provide the following kinds of data values for a gauge:

- 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 Property Inspector. It can also be specified through data controls or through the `tabularData` attribute of the `dvt: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 Property Inspector for the `dvt:gauge` tag. For more information, see Section 25.4.4, "How to Add Thresholds to Gauges."
- 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 Property Inspector. For more information, see Section 25.4.4, "How to Add Thresholds to Gauges."

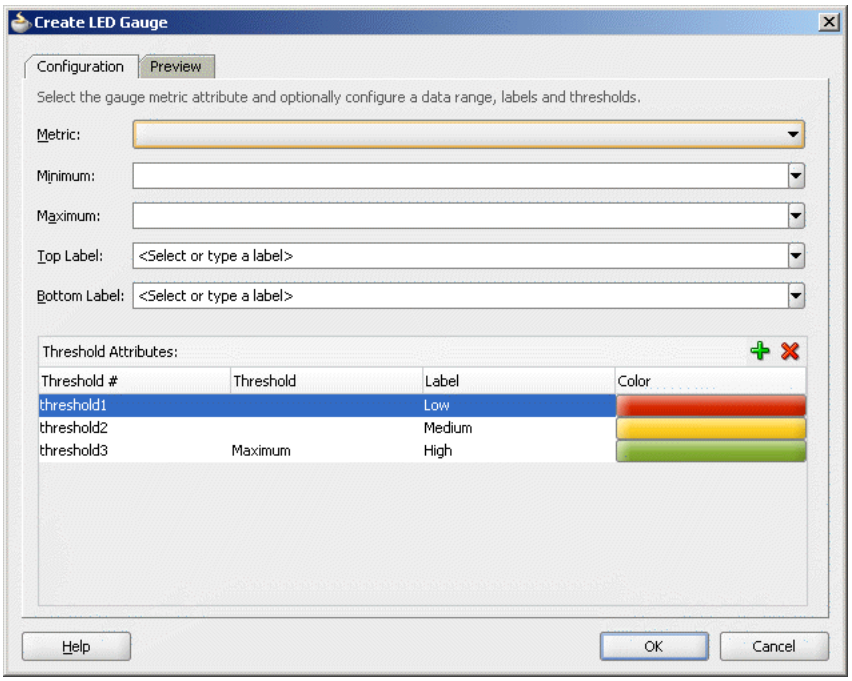
The only required data element is the metric value. All other data values are optional.

25.3 Creating a Gauge

You can use any of the following ways to supply data to a gauge component:

- ADF Data Controls: You declaratively create a databound gauge by dragging and dropping a data collection from the ADF Data Controls panel. Figure 25–8 shows the Create Gauge dialog where you configure the metric value, and optionally the minimum and maximum and threshold values for a gauge you are creating.

Figure 25–8 Create Gauge Dialog



For more information, see the "Creating Databound ADF Gauges" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- Tabular data: You can provide CSV (comma-separated value) data to a gauge through the `tabularData` attribute of the `dvt:gauge` tag.

25.3.1 Creating a Gauge Using Tabular Data

The process of creating a gauge from tabular data includes the following steps:

- Storing the data in a method in the gauge's managed bean.
- Creating a gauge that uses the data stored in the managed bean.

25.3.1.1 Storing Tabular Data for a Gauge in a Managed Bean

The `tabularData` attribute of a gauge component lets you specify a list of metric values that the gauge uses to create a grid and to populate itself. You can provide only the metric value through the `tabularData` attribute. Therefore, you must specify any desired thresholds and minimum or maximum values through the Property Inspector.

A gauge component displays rows and columns of gauges. The text that you specify as column labels appears in the top label of the gauges. The text that you specify as row labels appears in the bottom label of the gauges.

25.3.1.2 Structure of the List of Tabular Data

The list that contains the 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, in the grid, of the data value. This is generally a `String`.

- The second member (index 1) is the row label, in the grid, of the data value. This is generally a `String`.
- The third member (index 2) is the data value, which is usually `Double`.

Figure 25–9 has five columns: Quota, Sales, Margin, Costs, and Units. The example has three rows: London, Paris, and New York. This data produces a gauge set with five gauges in each row and lets you compare values such as sales across the three cities.

Figure 25–9 Comparison of Annual Results

	Quota	Sales	Margin	Costs	Unit
London	60	50	10	70	110
Paris	90	-100	40	80	120
New York	135	-150	150	-130	130

Example 25–2 shows code that creates the list of tabular data required for the gauge that compares annual results for three cities.

Example 25–2 Code to Create a List of Tabular Data for a Gauge

```
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[][]{
        {60, 90, 135},
        {50, -100, -150},
        {130, 140, 150},
        {70, 80, -130},
        {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;
}
```

25.3.2 How to Create a Gauge Using Tabular Data

Use the `tabularData` attribute of the gauge tag to reference the tabular data that is stored in a managed bean.

To create a gauge that uses tabular data from a managed bean:

1. In the ADF Data Visualizations page of the Component Palette, Gauge panel, drag and drop a **Gauge** onto the page.
2. In the Component Gallery, select the category, type, and quick-start layout style for the gauge that you are creating.
3. In the **Gauge Data** category of the Property Inspector, from the `tabularData` attribute dropdown menu, choose **Expression Builder**.

4. In the Expression Builder dialog, use the search box to locate the managed bean.
5. Expand the managed bean node and select the method that creates the list of tabular data.
6. 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 `dvt:gauge` tag.

25.3.3 What Happens When You Create a Gauge Using Tabular Data

When you create a gauge tag that is powered by data obtained from a list referenced in the `tabularData` attribute, the following results occur:

- A gauge is generated with a setting in its `tabularData` attribute. The settings for all other attributes for this gauge are provided by default.
- You have the option of changing the setting of the `gaugeType` attribute in the Property Inspector to `DIAL`, `LED`, `STATUSMETER`, or `VERTICALSTATUSMETER`.

25.3.4 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. By default, gauges display in Flash, but you can configure your application to use a specific image format by setting the following parameters:

- `oracle.adf.view.rich.dvt.DEFAULT_IMAGE_FORMAT`

  To use the HTML5 image format, add this parameter to the `web.xml` file and set it to `HTML5`. For more information, see Section A.2.3.23, "Graph and Gauge Image Format."

- `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 isn't 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 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

25.4 Customizing Gauge Type, Layout, and Appearance

Gauge components can be customized in the following ways:

- Change the gauge type
- Specify the layout of gauges in a gauge set
- Change a gauge size and style
- Add thresholds
- Format numbers and text
- Specify an N-degree dial gauge
- Customize gauge labels
- Customize indicators and tick marks
- Specifying transparency in gauges

25.4.1 How to Change the Type of the Gauge

You can change the type of a gauge using the `gaugeType` attribute of the `dvt:gauge` tag. The gauge type is reflected in the visual editor default gauge.

To change the type of a gauge:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the Property Inspector, choose a gauge type from the **GaugeType** attribute dropdown list. Valid values are `DIAL`, `LED`, `STATUSMETER`, or `VERTICALSTATUSMETER`.

25.4.2 How to Determine the Layout of Gauges in a Gauge Set

A single gauge can display one row of data bound to a gauge component. A gauge set displays a gauge for each row in multiple rows of data in a data collection.

You can specify the location of gauges within a gauge set by specifying values for attributes in the `dvt:gauge` tag.

To specify the layout of gauges in a gauge set:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the Property Inspector, select the **Common** attributes category.
3. To determine the number of columns of gauges that will appear in a gauge set, specify a value for the `gaugeSetColumnCount` attribute.

   A setting of zero causes all gauges to appear in a single row. Any positive integer determines the exact number of columns in which the gauges are displayed. A setting of -1 causes the number of columns to be determined automatically from the data source.
4. To determine the placement of gauges in columns, specify a value for the `gaugeSetDirection` attribute.

   If you select **GSD_ACROSS**, then the default layout of the gauges is used and the gauges appear from left to right, then top to bottom. If you select **GSD_DOWN**, the layout of the gauges is from top to bottom, then left to right.

5. To control the alignment of gauges within a gauge set, specify a value for the `gaugeSetAlignment` attribute.

   This attribute defaults to the setting **GSA_NONE**, which divides the available space equally among the gauges in the gauge set. Other options use the available space and optimal gauge size to allow for alignment towards the left or right and the top or bottom within the gauge set. You can also select **GSA_CENTER** to center the gauges within the gauge set.

25.4.3 Changing Gauge Size and Style

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. You can also control the style sheet used by a gauge. These two aspects of a gauge are interrelated in that they share the use of the gauge `inlineStyle` attribute.

25.4.3.1 Specifying the Size of a Gauge at Initial Display

You can specify the initial size of a gauge by setting values for attributes of the `dvt:gauge` tag. If you do not also provide for dynamic resizing of the gauge, then the initial size becomes the only display size for the gauge.

To specify the size of a gauge at its initial display:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the **Style** attributes category of the Property Inspector, enter a value for the `InlineStyle` attribute of the `dvt:gauge` tag. For example:

   ```
   inlineStyle="width:200px;height:200px"
   ```

25.4.3.2 Providing Dynamic Resizing of a Gauge

You must enter values in each of two attributes of the `dvt:gauge` tag to allow for a gauge 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 gauge component that fills an area across different browser window sizes.

To allow dynamic resizing of a gauge:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the **Behavior** attributes category of the Property Inspector for the `DynamicResize` attribute, select the value **DYNAMIC_SIZE**.
3. In the **Style** attributes category of the Property Inspector, for the `InlineStyle` attribute, enter a fixed number of pixels or a relative percent for both width and height.

   For example, to create a gauge that fills its container's width and has a height of 200 pixels, use the following setting for the `inlineStyle` attribute: `"width:100%;height:200px;"`

25.4.3.3 Using a Custom Style Class for a Gauge

You have the option of specifying a custom style class for use with a gauge. However, you must specify width and height in the `inlineStyle` attribute.

To specify a custom style class for a gauge:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the **Style** attributes category of the Property Inspector, for the `StyleClass` attribute, select Edit from the Property menu choices, and select the CSS style class to use for this gauge.
3. In the `InlineStyle` attribute, enter a fixed number of pixels or a relative percent for both width and height.

   For example, to create a gauge that fills its container's width and has a height of 200 pixels, use the following setting for the `inlineStyle` attribute: `"width:100%;height:200px;"`.

25.4.4 How to Add Thresholds to Gauges

Thresholds are 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. After the gauge is created, you can also insert a `dvt:thresholdSet` tag and individual `dvt:threshold` tags to create static thresholds. If threshold values are supplied in both the data collection and in threshold tags, then the gauge honors the values in the threshold tags.

25.4.4.1 Adding Static Thresholds to Gauges

You can create an indefinite number of thresholds in a gauge. Each threshold is represented by a single `dvt:threshold` tag. One `dvt:thresholdSet` tag must wrap all the threshold tags.

To add static thresholds to a gauge:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Threshold Set**.

   You do not need to specify values for attributes on the `dvt:thresholdSet` tag.
2. Right-click the **dvt:thresholdSet** node and choose **Insert inside dvt:thresholdSet** > **threshold**.
3. In the Property Inspector, enter values for the attributes that you want to customize for this threshold.

   You have the option of entering a specific fill color and border color for the section of the gauge related to the threshold. You can also identify the maximum value for the threshold and any text that you want to display in the legend to identify the threshold.

   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 threshold tag for the final threshold.
4. Repeat Step 2 and Step 3 to create each threshold in the gauge from the lowest minimum value to the highest maximum value.

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.

25.4.5 How to Format Numeric Values in Gauges

For gauges, the `dvt:metricLabel` and `dvt:tickLabel` tags may require numeric formatting.

25.4.5.1 Formatting the Numeric Value in a Gauge Metric Label

The metric label tag 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.

You can also use the `af:convertNumber` tag to specify formatting for numeric values in the metric label. For example, the `af:convertNumber` tag lets you format data values as currency or display positive or negative signs.

To format numbers in a gauge metric label:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **metricLabel**.
2. If you want to display the metric value as a percentage rather than as a value, then set the `NumberType` attribute of the `dvt:metricLabel` tag to **NT_PERCENT**.
3. If you want to specify additional formatting for the number in the metric label, do the following:
   a. Right-click the **metricLabel** node and choose **Insert inside dvt:metricLabel** > **Convert Number**.
   b. In the Property Inspector, specify values in the attributes of the `af:convertNumber` tag to produce additional formatting.

The procedure for formatting numbers in gauge tick labels is similar to that of formatting numbers in the metric label except that you insert the `dvt:tickLabel` tag as a child of the gauge.

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.

25.4.6 What Happens When You Format the Numbers in a Gauge Metric Label

When you add a metric label and number formatting to a gauge, XML code is generated. Example 25–3 shows a sample of the XML code that is generated.

Example 25–3 XML Code Generated When Formatting a Number in a Metric Label

```
<dvt:gauge>
  <dvt:metricLabel position="LP_BELOW_GAUGE" numberType="NT_PERCENT">
    <af:convertNumber type="percent"/>
  </dvt:metricLabel>
</dvt:gauge>
```

25.4.7 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`.

25.4.8 How to Format Text in Gauges

You can format text in any of the following gauge tags that represent titles and labels in a gauge:

- `dvt:bottomLabel`
- `dvt:gaugeMetricLabel`
- `dvt:gaugeLegendText`
- `dvt:gaugeLegendTitle`
- `dvt:tickLabel`
- `dvt:topLabel`

The procedure for formatting text in gauge labels and titles is similar except that you insert the appropriate child tag that represents the gauge label or title. For example, you can use a `dvt:gaugeFont` child tag to a `dvt:metricLabel` tag to specify gauge metric label font size, color, and if the text should be bold or italic.

To format text in a gauge metric label:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **metricLabel**.
2. Right-click the **metricLabel** node and choose **Insert inside dvt:metricLabel** > **Font**.
3. In the Property Inspector, specify values in the attributes of the `dvt:gaugeFont` tag to produce the desired formatting.

When you format text in a gauge metric label using the `gaugeFont` tag, XML code is generated. Example 25–4 shows a sample of the XML code that is generated.

Example 25–4 XML Code Generated When You Format Text in a Gauge Metric Label

```
<dvt:gauge>
  <dvt:metricLabel>
     <dvt:gaugeFont name="Tahoma" size="11" color="#3C3C3C" bold="true"/>
  </dvt:metricLabel>
</dvt:gauge>
```

25.4.9 How to Specify an N-Degree Dial

You can specify a gauge that sweeps through angles other than the standard 220-degree arc in a dial gauge. Set the `angleExtent` attribute to specify the range of degrees in the gauge.

For example, to create a 270 degree dial gauge, set the `angleExtent` attribute as follows: `<dvt:gauge angleExtent="270"/>`.

25.4.10 How to Customize Gauge Labels

You can control the positioning of gauge labels. You can also control the colors and borders of the gauge label frames.

25.4.10.1 Controlling the Position of Gauge Labels

You can specify whether you want labels to appear outside or inside a gauge by using the `position` attribute of the appropriate label tag. The following label tags are available as child tags of `dvt:gauge`:

- `dvt:bottomLabel`
- `dvt:metricLabel`
- `dvt:topLabel`

The procedure for controlling the position of gauge labels is similar except that you insert the appropriate child tag that represents the gauge label. For example, you can use the `dvt:bottomLabel` child tag to position the gauge and specify label text.

To specify the position of the bottom label:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Bottom Label**.
2. In the Property Inspector, for the `position` attribute, select the desired location of the label.
3. In the `text` attribute, enter the text that you want the label to display.

25.4.10.2 Customizing the Colors and Borders of Gauge Labels

You can control the fill color and border color of the frames for the top label and the bottom label. The `dvt:upperLabelFrame` and `dvt:lowerLabelFrame` gauge child tags serve as frames for these labels.

To customize the color and border of the upper label frame:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Upper Label Frame**.
2. In the Property Inspector, select the desired colors for the `borderColor` attribute and the `fillColor` attribute.

Use a similar procedure to customize the color and border of the bottom label frame using the `dvt:bottomLabel` tag as a child of the gauge node.

25.4.11 How to Customize Indicators and Tick Marks

There are a variety of options available for customizing the indicators of gauges and the location and labeling of tick marks.

25.4.11.1 Controlling the Appearance of Gauge Indicators

The following gauge child tags are available to customize the indicator of a gauge:

- `dvt:indicator`: Specifies the visual properties of the dial gauge indicator needle or the status meter bar. Includes the following attributes:
  - `borderColor`: Specifies the color of the border of the indicator.
  - `fillColor`: Specifies the color of the fill for the indicator.
  - `type`: Identifies the kind of indicator: a line indicator, a fill indicator, or a needle indicator.
  - `useThresholdFillColor`: Determines whether the color of the threshold area in which the indicator falls should override the specified color of the indicator.
- `dvt:indicatorBar`: Contains the fill properties of the inner rectangle (bar) of a status meter gauge.
- `dvt:indicatorBase`: Contains the fill properties of the circular base of a line and needle style indicator of a dial gauge.

To customize the appearance of gauge indicators:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Indicator**.
2. In the Property Inspector, specify values for the desired attributes.
3. If you want to customize the fill attributes of the inner bar on a status meter gauge, in the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Indicator Bar**.
4. In the Property Inspector, specify values for the desired attributes.
5. If you want to customize the circular base of a line style indicator on a dial gauge, in the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Indicator Base**.
6. In the Property Inspector, specify values for the desired attributes.

25.4.11.2 Specifying Tick Marks and Labels

The following gauge child tags are available to customize tick marks and tick labels for a gauge:

- `dvt:tickMark`: Specifies the display, spacing, and color of major and minor tick marks. Only major tick marks can include value labels. Includes 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.
- `dvt:tickLabel`: Identifies major tick marks that will have labels, the location of the labels (interior or exterior of the gauge), and the format for numbers displayed in the tick labels.

To customize the tick marks and tick labels of a gauge:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Tick Mark**.
2. In the Property Inspector, specify values for the desired attributes.
3. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Tick Label**.
4. In the Property Inspector, specify values for the desired attributes.

25.4.11.3 Creating Exterior Tick Labels

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.

To create interior tick labels on a gauge:

1. In the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Tick Mark**.
2. In the Property Inspector, select **TLP_POSITION** for the `Position` attribute.

25.4.12 Specifying Transparency for Parts of a Gauge

You can specify that various parts of a gauge show transparent colors by setting the `borderColor` and `fillColor` attributes on the gauge child tags related to these parts of the gauge. 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 tag that supports `borderColor` or `fillColor` attributes can be set to transparency. The following list are examples of parts of the gauge that support transparency:

- Gauge background: Use the `dvt:gaugeBackground` tag.
- Gauge gauge frame: Use the `dvt:gaugeFrame` tag.
- Gauge plot area: Use the `dvt:gaugePlotArea` tag.
- Gauge legend area: Use the `dvt:gaugeLegendArea` tag.

25.5 Adding Gauge Special Effects and Animation

These gauge features are used less frequently than the common gauge features. These special features include applying gradient effects to parts of a gauge, adding

interactivity to gauges, animating gauges, and taking advantage of the gauge support for active data.

25.5.1 How to Use Gradient Special Effects in 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 subcomponent of a gauge that supports special effects.

You can define gradient special effects for the following subcomponents of a gauge:

- Gauge background: Uses the `dvt:gaugeBackground` tag.
- Gauge set background: Uses the `dvt:gaugeSetBackground` tag.
- Gauge plot area: Uses the `dvt:gaugePlotArea` tag.
- Gauge frame: `Uses the dvt:gaugeFrame` tag.
- Gauge legend area: Uses the `dvt:gaugeLegendArea` tag.
- Lower label frame: Uses the `dvt:lowerLabelFrame` tag.
- Upper label frame: Uses the `dvt:upperLabelFrame` tag.
- Indicator: Uses the `dvt:indicator` tag.
- Indicator bar: Uses the `dvt:indicatorBar` tag.
- Indicator base: Uses the `dvt:indicatorBase` tag.
- Threshold: Uses the `dvt:threshold` tag.

The approach that you use to define gradient special effects is identical for each part of the gauge that supports these effects.

25.5.1.1 Adding Gradient Special Effects to a Gauge

For each subcomponent of a gauge to which you want to add special effects, you must insert a `dvt:specialEffects` tag as a child tag of the subcomponent. For example, if you want to add a gradient to the background of a gauge, then you would create one `dvt:specialEffects` tag that is a child of the `dvt:background` tag. You must also set the `dvt:specialEffects` tag `fillType` property to `FT_GRADIENT`.

Then, optionally if you want to control the rate of change for the fill color of the subcomponent, you would add 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 entered as child tags of the single `dvt:specialEffects` tag.

Before you begin:

If you have not inserted a `dvt:gaugeBackground` tag as a child of the gauge, in the Structure window, right-click the gauge node and choose **Insert inside dvt:gauge** > **ADF Data Visualization** > **Gauge Background**.

To add a gradient special effect to the background of a gauge:

1. In the Structure window, right-click the gauge background node and choose **Insert inside dvt:gaugeBackground** > **Special Effects**.
2. Use the Property Inspector to enter values for the attributes of the `dvt:specialEffects` tag:

a. For `fillType` attribute, select `FT_GRADIENT`.

b. For `gradientDirection` attribute, select the direction of change that you want to use for the gradient fill.

c. For the `numStops` attribute, enter the number of stops to use for the gradient.

3. Optionally, in the Structure window, right-click the special effects node and choose **Insert within dvt:specialEffects** > **dvt:gradientStopStyle** if you want to control the color and rate of change for each gradient stop.

4. Use the Property Inspector to enter values for the attributes of the `dvt:gradientStopStyle` tag:

a. For the `stopIndex` attribute, enter a zero-based integer as an index within the `dvt:gradientStopStyle` tags that are included within the `dvt:specialEffects` tag.

b. For the `gradientStopColor` attribute, enter the color that you want to use at this specific point along the gradient.

c. For the `gradientStopPosition` attribute, 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.

5. Repeat Step 3 and Step 4 for each gradient stop that you want to specify.

25.5.2 What Happens When You Add a Gradient Special Effect to a Gauge

When you add a gradient fill to the background of a gauge and specify two stops, XML code is generated. Example 25–5 shows the XML code that is generated.

Example 25–5 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>
```

25.5.3 How to Add Interactivity to Gauges

You can specify interactivity properties on subcomponents of a gauge using one or more `dvt:shapeAttributes` tags wrapped in a `dvt:shapeAttributesSet` tag. The interactivity provides a connection between a specific subcomponent and an HTML attribute or a JavaScript event. Each `dvt:shapeAttributes` tag must contain a subcomponent and at least one attribute in order to be functional.

For example, Example 25–6 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 25–6 Sample Code for Gauge shapeAttributes Tag

```
<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 backing bean method to return the value of the attribute. Example 25–7 shows sample code for referencing a backing bean and Example 25–8 shows the backing bean sample code.

Example 25–7 Gauge shapeAttributes Tag Referencing a Backing 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>
```

Example 25–8 Sample Backing 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\";"); }
```

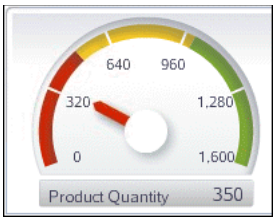
The following gauge subcomponents support the `dvt:shapeAttributes` tag:

- `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

25.5.4 How to Animate Gauges

You can animate gauges (not gauge sets) to show changes in data, for example, a dial gauge indicator can change color when a data value increases or decreases. Figure 25–10 shows a dial gauge with the dial indicator animated to display the data change at each threshold level.

Figure 25–10 Animated Dial Gauge



The attributes for setting animation effects on gauges are:

- `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 25.5.5, "How to Animate Gauges with 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.

25.5.5 How to Animate Gauges with 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.

Before you begin:

In order to use the Active Data Service, you must:

- Have a data source that publishes events when data is changed
- 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 the "Using the Active Data Service" chapter in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

25.5.5.1 Configuring Gauge Components to Display Active Data

You configure a databound gauge to display active data by setting a value on the binding element in the corresponding page definition file.

To configure a gauge to display active data:

1. In the Structure window, right-click the gauge node, and choose **Go To Properties**.

2. In the Property Inspector, enter a unique value in the ID field.

   If you do not select an identifier, one will be entered for you.

3. Open the page's associated page definition file.

4. In the Structure window for the page definition file, select the node that represents the attribute binding for the component. In the Property Inspector, select **Push** for the `ChangeEventPolicy` attribute.

25.5.5.2 Adding Animation to Gauges

After configuring the gauge component to display active data, set animation effects using the attributes defined in Section 25.5.4, "How to Animate Gauges."

25.6 Using Custom Shapes in Gauges

You can directly specify the graphics for a gauge to create custom gauge shapes. The `customShapesPath` attribute is set to point to the vector graphics file that is processed into graphics used for output. JDeveloper also provides a set of custom shape styles accessible by using the `customShapesPath` attribute.

25.6.1 How to Create 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 25–1 shows the gauge component IDs and their descriptions.

Table 25–1 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`.

Table 25–1 (Cont.) Gauge Component IDs for Custom Shapes

ID	Description
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 25–2 shows the metadata IDs and the descriptions used for internal calculations, not rendered in the gauge.

Table 25–2 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.

Table 25–2 (Cont.) Metadata IDs for Custom Shapes

ID	Description
`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 25–9 shows a sample SVG file used to specify custom shapes for the components of a gauge.

Example 25–9 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>
```

25.6.2 How to Use a Custom Shapes File

After creating the SVG file to be used to specify the custom shapes for your gauge, set the `customShapesPath` attribute to point to the file.

To specify a custom shapes file:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the **Appearance** attributes category of the Property Inspector, for the `CustomShapesPath` attribute, enter the path to the custom shapes file. For example, `customShapesPath="/path/customShapesFile.svg"`.

25.6.3 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 in JDeveloper 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.

25.6.4 How to Set Custom Shapes Styles

In addition to the ability to specify custom shapes for gauges, there are a set of prebuilt custom shapes styles for use with the gauge components. The available styles are:

- Rounded rectangle
- Full circle
- Beveled circle

Figure 25–11 shows a dial gauge displayed with each of the custom shapes styles applied.

Figure 25–11 Dial Gauges with Custom Shapes Styles

To apply a custom shapes style to a gauge:

1. In the Structure window, right-click the **dvt:gauge** node and choose **Go to Properties**.
2. In the **Appearance** attributes category of the Property Inspector, select the custom shapes style from the `CustomShapesPath` attribute dropdown list.

26

Using ADF Pivot Table Components

This chapter describes how to use a databound ADF pivot table component to display data, and provides the options for pivot table customization.

This chapter includes the following sections:

For information about the data binding of ADF pivot tables, see the "Creating Databound ADF Pivot Tables" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

26.1 Introduction to the ADF Pivot Table Component

The ADF pivot table component displays a grid of data with rows and columns. Similar to spreadsheets, this component provides the option of automatically generating subtotals and totals for grid data. The pivot table lets you *pivot* or move data labels and the associated data layer from one row or column edge to another to obtain different views of your data, supporting interactive analysis.

The power of the pivot table's interactive capability is based in its display of multiple nested attributes on row and column headers. You can dynamically change the layout of these attributes using drag-and-drop operations.

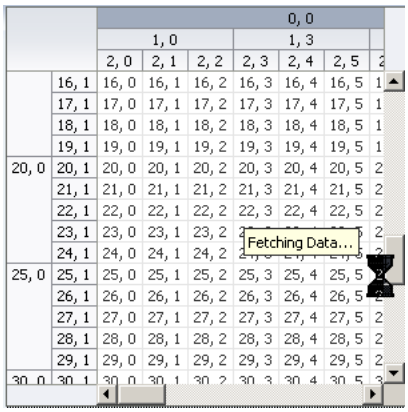
Figure 26–1 shows a pivot table with multiple attributes nested on its rows and columns.

Figure 26–1 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

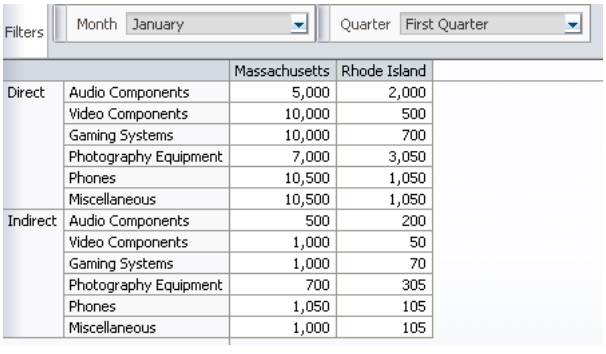
Pivot tables support on-demand data scrolling for large data sets. Only the data being viewed in the pivot table is loaded. As the user scrolls vertically or horizontally, data is fetched or discarded to fill the new pivot table view. Figure 26–2 shows a pivot table with a large data set using on-demand data scrolling.

Figure 26–2 On-Demand Data Scrolling in a Pivot Table



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 other 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 26–3 shows a pivot filter bar for a pivot table.

Figure 26–3 Pivot Filter Bar Component



26.1.1 Pivot Table Elements and Terminology

The following list of pivot table terms uses Figure 26–1 as a Sales Pivot Table sample in its descriptions of terms:

- Edges: The axes in pivot tables, including:

  - Row edge: The vertical axis to the left of the body of the pivot table. In Figure 26–1, the row edge contains two layers, Year and Product, and each row in the pivot table represents the combination of a particular year and a particular product.
  - Column edge: The horizontal axis above the body of the pivot table. In Figure 26–1, the column edge contains three layers, Measure, Channel, and Geography, and each column in the pivot table represents the combination of a particular measure value (Sales or Units), a particular channel indicator (All Channels), and a particular geographic location (World or Boston).
  - 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 26–1, the following three layers appear in the column edge: Measure, Channel, and Geography. The following two layers appear in the row edge: Year 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.
- 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 20,000.000.
- QDR (Qualified Data Reference): A fully qualified data reference to a row, a column, or an individual cell. For example, in Figure 26–1, the QDR for the first data cell in the pivot table must provide the following information:
  - Year=2007
  - Product=Tents
  - Measure=Sales
  - Channel=All Channels
  - Geography=World

26.2 Understanding Data Requirements for a Pivot Table

The pivot table component uses a model to display and interact with data. The specific model class used is `oracle.adf.view.faces.bi.model.DataModel`.

You can use any row set (flat file) data collection to supply data to a pivot table. During the data binding operation, you have the opportunity to drag each data element to the desired location on the row edge or column edge of the pivot table in the data binding dialog.

During data binding, you also have the option of specifying subtotals and totals for pivot table rows and columns, specifying drill operations at runtime, defining how to aggregate duplicate records, and setting up initial sort criteria.

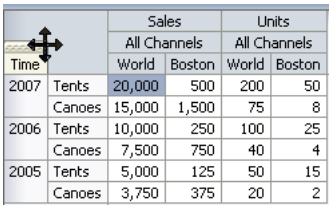
For information about the data binding of ADF pivot tables, see the "Creating Databound ADF Pivot Tables" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

26.3 Pivoting Layers

You can drag any layer in a pivot table to a different location on the same edge or to a different edge. 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 26–4.

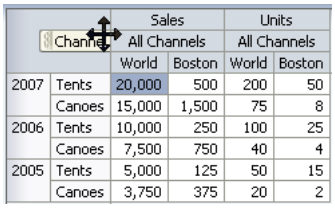
Figure 26–4 Display of Pivot Handle on the Row Edge



		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 26–5.

Figure 26–5 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 26–4, 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 26–6.

Figure 26–6 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.

To customize pivoting in a pivot table:

1. In the Structure window, right-click the **dvt:pivotTable** component and choose **Go to Properties**.
2. Optionally, in the **Appearance** category of the Property Inspector, in the `PivotLabelVisible` field, select **false** from the dropdown list to disable the display of the label in the pivot handle.
3. Optionally, in the **Behavior** category of the Property Inspector, in the `PivotEnabled` field, select **false** from the dropdown list to disable the pivoting.

26.4 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 26–1 shows sample code for getting the currently selected header cells.

Example 26–1 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, selection in a data cell highlights the cell, as shown in Figure 26–7.

Figure 26–7 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

Editable data cells are opened for editing by double-clicking the cell or selecting the cell and pressing F2. Data cells selected for direct editing are displayed as shown in Figure 26–8.

Figure 26–8 Data Cell Open for Direct Editing

				Sales	Units	Enter between 2 and 6 characters.				Link	Size
2007	Tents	All Channels	World	20000.0	200.(	☑ Item Available	33.0	red	33.0	Main-link	L
			Boston	500.0	50.0	☐ Item Available	66.0	coffee	66.0	Main-link	S
2008	Jacket	All Channels	Cambridge	500.0	50.0	☑ Item Available	44.0	milk	44.0	Sub-link	M

Data cells selected for dropdown list editing are displayed as shown in Figure 26–9.

Figure 26–9 Data Cell Open for Dropdown List Editing

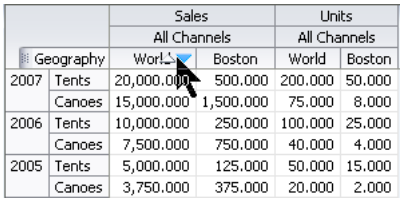
				Sales	Units	Available	Price	Color	Weight	Link	Size
2007	Tents	All Channels	World	20000.0	200.0	☑ Item Available	33.0	re	33.0	Main-link	L
			Boston	500.0	50.0	☐ Item Available	66.0	red coffee milk	6.0	Main-link	S
2008	Jacket	All Channels	Cambridge	500.0	50.0	☑ Item Available	44.0		4.0	Sub-link	M

For more information about enabling data cell editing, see Section 26.10, "Pivot Table Data Cell Stamping and Editing."

26.5 Sorting in a Pivot Table

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 cursor over the innermost layer of the column header. By default, the `sortMode` attribute of the `dvt:pivotTable` component is set to `grouped`, effectively sorting the data grouped by the row edge outermost layer. Figure 26–10 shows the sort icons in the World Sales column of the pivot table, where the data is grouped by the Year row edge outermost layers.

Figure 26–10 Ascending and Descending Sorting Icons in a Pivot Table



		Sales		Units	
		All Channels		All Channels	
Geography		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

26.6 Sizing in a Pivot Table

When you create a pivot table, default settings determine the overall size of that pivot table. The pivot table also automatically sizes rows, columns, and layers within the space allowed for the overall size. You have the option of changing the overall size of the pivot table, resizing rows and columns, and resizing layers.

26.6.1 How to Set the Overall Size of a Pivot Table

The default size of a pivot table is a width of 300 pixels and a height of 300 pixels. Instead of entering pixels for width and height, you have the option of specifying a percentage value for width, height, or both. This percentage value refers to the portion of the page that you want the pivot table to use.

To customize the default settings of a pivot table:

1. In the visual editor, display the page that contains the pivot table.
2. Click **Source** to display the XML code on the JSPX page.
3. Enter the following code for the `inlineStyle` attribute of the `pivotTable` tag, where `value1` is an integer with the unit type for the width of the pivot table and `value2` is an integer with the unit type for the height of the pivot table: `inlineStyle="width:value1;height:value2"`.

Example 26–2 shows the setting of the `inlineStyle` attribute that specifies the width of the table as 50 percent of the page size and the height of the table as 400 pixels.

Example 26–2 XML Code for Customizing Pivot Table Size

```
<dvt:pivotTable
.
.
.
   inlineStyle="width:50%;height:400px">
</dvt:pivotTable>
```

26.6.2 How to Resize Rows, Columns, and Layers

The pivot table autosizes rows, columns, and layers 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.

To resize rows, columns, and layers at runtime:

1. If you want to resize a row, do the following:
   a. Position the cursor in the row header on the separator between the row you want to resize and the next row.
   b. When the cursor changes to a double-sided arrow, click and drag the row separator to the desired location.
2. If you want to resize a column, do the following:
   a. Position the cursor in the column header on the separator between the column you want to resize and the next column.
   b. When the cursor changes to a double-sided arrow, click and drag the column separator to the desired location.
3. If you want to resize a layer, do the following:
   a. Position the cursor in the row or column header on the separator between the layer you want to resize and the next layer.
   b. When the cursor changes to a double-sided arrow, click and drag the layer separator to the desired location.

26.6.3 What You May Need to Know About Resizing Rows, Columns, and Layers

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.

26.7 Updating Pivot Tables with Partial Page Rendering

You can update pivot tables, for example, to display the totals in a pivot table when triggered by a checkbox, by using partial page rendering (PPR). PPR allows only certain components on a page to be rerendered without the need to refresh the entire page. For more information about PPR, see Chapter 7.1, "Introduction to Partial Page Rendering."

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 26–3 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.

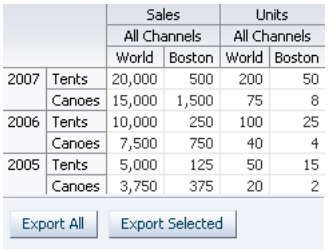
Example 26–3 Partial Update of a Pivot Table

```
<dvt:pivotTable id="goodPT"
    value="#{richPivotTableModel.dataModel}"
    partialTriggers="showTotals"/>

  <af:selectBooleanCheckbox id="showTotals" autoSubmit="true" label="Show Totals"
    value="#{richPivotTableModel.totalsEnabled}"/>
```

26.8 Exporting from a Pivot Table

You can export the data from a pivot table to a Microsoft Excel spreadsheet. Create an action source, such as a command button or command link, and add a `dvt:exportPivotTableData` component and associate it with the data you wish to export. You can configure the `dvt:exportPivotTableData` component so that the entire pivot table will be exported, or so that only the rows selected by the user will be exported. For example, Figure 26–11 shows a pivot table that includes command button components that allow users to export the data to an Excel spreadsheet.

Figure 26–11 Pivot Table with Export to Excel Command Buttons



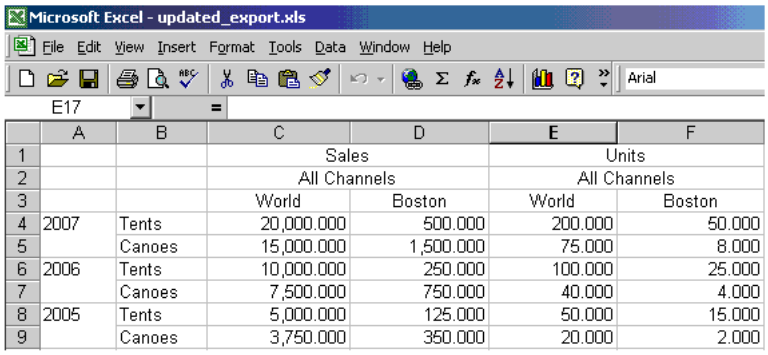
When the user clicks the command button, by default all the rows and columns are exported in an Excel format written to the file specified in the `filename` attribute of the tag. Alternatively, you can configure the `dvt:exportPivotTableData` component so that only the rows the user selects are exported, by setting the `exportedData` attribute to `selected`. Example 26–4 shows the sample code for the **Export to Excel** command button.

Example 26–4 Sample Code for Export to Excel Command Button

```
<af:commandButton text="Export To Excel" immediate="true">
  <dvt:exportPivotTableData type="excelHTML" exportedId="goodPT"
           filename="updated_export.xls" title="PivotTable export"/>
</af:commandButton>
```

Figure 26–12 shows the resulting Excel spreadsheet.

Figure 26–12 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.

26.9 Customizing the Cell Content of a Pivot Table

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 `dvt:pivotTable` tag:

- 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.

26.9.1 How to Create 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`.

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.

26.9.2 How to Construct a CellFormat Object

An instance of a `CellFormat` object lets you specify the following arguments:

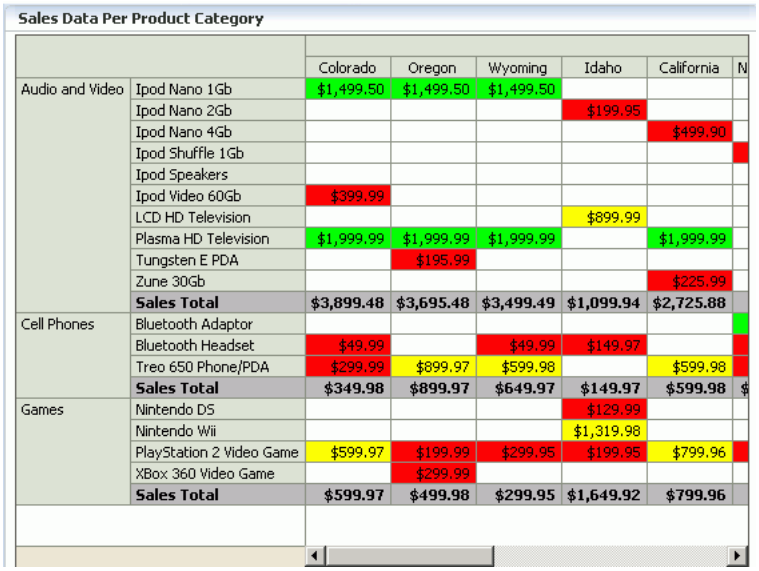
- Converter: An instance of `javax.faces.convert.Converter`, which is used to perform number, date, or text formatting of a raw value in a cell.
- CSS style: Used to change the CSS style of a cell. For example, you might use this argument to change the background color of a cell.
- CSS text style: Used 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.

26.9.3 How to Change Format and Text Styles

You can apply formatting and text styles to emphasize aspects of the data displayed in the pivot table. Figure 26–13 shows a pivot table with sales totals generated for products and for product categories. In the rows that contain totals, this pivot table displays bold text (a text style change) against a shaded background (a style change). These changes show 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 26.9.4, "How to Create Stoplight and Conditional Formatting in a Pivot Table."

Figure 26–13 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 26–5 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 26–6.

Example 26–5 Sample Code to Change Style and Text 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.setTextStyle("font-weight:bold")
       cellFormat.setStyle("background-color:#C0C0C0");
      }
     else if (product != null && product.toString().equals("Sales Total")
      {
       cellFormat.setTextStyle("font-weight:bold");
       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",
                              "font-weight:bold");
      }
    }
    return null;
  }
```

26.9.4 How to Create Stoplight and Conditional Formatting in a Pivot Table

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 follow:

- 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 26–13 shows data cells with stoplight formatting for minimum, acceptable, and below standards sales for States.

Example 26–6 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 26–5 (which addresses rows with totals) with the code for stoplight and conditional formatting.

Example 26–6 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;
}
```

26.10 Pivot Table Data Cell Stamping and Editing

The content in a pivot table data cell can be stamped using the `dvt:dataCell` child component to place a read-only or input component in each data cell. When you use stamping, child components are not created for every data cell in a pivot table. Rather, the content of the `dvt:dataCell` component is repeatedly rendered, or stamped, once per data attribute, such as the rows in a pivot table. Only certain types of components are supported, including all components with no activity and most components that implement the `EditableValueHolder` or `ActionSource` interfaces. You can also use stamping to specify custom CSS styles for the data cell.

Each time a child component is stamped, the data for the current cell is copied into a `var` property used by the data cell component in an EL Expression. Once the pivot table has completed rendering, the `var` property is removed, or reverted back to its previous value.

Data cell editing is enabled by using an input component as the child component of `dvt:dataCell`. At runtime you can open the cell for editing by double-clicking the cell in the pivot table, or by selecting the cell and pressing F2.

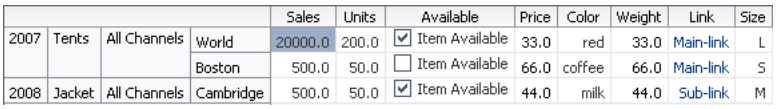
Example 26–7 shows sample code for data cell stamping.

Example 26–7 Data Cell Stamping Sample Code

```
<dvt:pivotTable var="cellData" varStatus="cellStatus">
  <dvt:dataCell>
    <af:switcher defaultFacet="default"
facetName="#{cellStatus.members.MeasDim.value}">
      <f:facet name="Sales">
        <af:inputText value="#{cellData.dataValue}"   />
      </f:facet>
      <f:facet name="Weight">
        <af:outputText value="#{cellData.dataValue}"
inlineStyle="#{cellStatus.cellFormat.textStyle}"/>
      </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="default">
        <af:outputText  value="#{cellData.dataValue}"   />
      </f:facet>
    </af:switcher>
  </dvt:dataCell>
</dvt:pivotTable>
```

Figure 26–14 shows the resulting pivot table.

Figure 26–14 Pivot Table Using Data Cell Stamping



				Sales	Units	Available	Price	Color	Weight	Link	Size
2007	Tents	All Channels	World	20000.0	200.0	☑ Item Available	33.0	red	33.0	Main-link	L
			Boston	500.0	50.0	☐ Item Available	66.0	coffee	66.0	Main-link	S
2008	Jacket	All Channels	Cambridge	500.0	50.0	☑ Item Available	44.0	milk	44.0	Sub-link	M

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 row set based data control is transformed into a cube and therefore cannot support writeback operations.

26.10.1 How to Specify Custom Images for Data Cells

With data cell stamping you can use the `dvt:dataCell` tag to specify a custom image for a data cell using `af:image`, `af:icon`, or `af:commandImageLink` as a child tag. Example 26–8 shows sample code for using an `af:commandImageLink` as a custom image in a pivot table data cell.

Example 26–8 Using a Custom Image for a Data Cell

```
<dvt:pivotTable var="cellData" varStatus="cellStatus">

  <!-- This is the default data cell that will be used for all data attributes-->
  <dvt:dataCell>
    <af:commandImageLink text="Go"
                         icon="/images/go.gif"
     actionListener="#{pivotTableBean.imageLinkClick}"/>
    <af:outputText  value="#{cellData.dataValue}"   />
  </dvt:dataCell>
</dvt:pivotTable>
```

Actions associated with the image are handled through a registered listener, `actionListener`. In a bean class you specify the method to be called when the image link is clicked, for example:

```
public void imageLinkClick (javax.faces.event.ActionEvent.action)
```

26.10.2 How to Specify Images, Icons, Links, and Read-Only Content in Header Cells

In the same way that you use stamping in data cells, you can customize the content in header cells using the `dvt:headerCell` child component to place a read-only component in each header cell. Only read-only components or noneditable components are supported, including `af:outputText`, `af:image`, `af:icon`, `af:commandImageLink`, and `af:commandLink`.

Example 26–9 shows sample code for using an `af:commandImageLink` as a custom image and `af:icon` as a custom icon in pivot table header cells.

Example 26–9 Using Custom Components in Header Cells

```
<dvt:pivotTable id="goodPT"
                  binding="#{pivotTableHeaderCellDemo.pivotTable}"
                  contentDelivery="immediate"
                  value="#{pivotTableHeaderCellDemo.dataModel}"
                  var="cellData"
                  varStatus="cellStatus">
<!-- header cell components -->
```

```
<dvt:headerCell>
   <af:switcher defaultFacet="default" facetName="#{cellData.layerName}">
     <f:facet name="Geography">
       <af:outputText  value="#{cellData.label}"
          shortDesc="#{cellData.label}"/>
       <af:icon name="info" shortDesc="#{cellData.indent}" />
     </f:facet>
       <f:facet name="Channel">
       <af:outputText  value="#{cellData.label}"  />
       <af:commandImageLink shortDesc="Sample commandImageLink"
         icon="/resources/images/pivotTableCSVDemo/smily-normal.gif"
         hoverIcon="/resources/images/pivotTableCSVDemo/smily-glasses.gif"
         />
       <af:commandButton text="Go to Tag Guide page" immediate="true"
         action="guide" />
      </f:facet>

      <f:facet name="Product">
        <af:outputText  value="#{cellData.label}" />
        <af:commandButton text="Go to Tag Guide page" immediate="true"
         action="guide" />
      </f:facet>

      <f:facet name ="default">
        <af:commandLink text="#{cellData.label}"
         immediate="true" action="guide"/>
      </f:facet>
    </af:switcher>
  </dvt:headerCell>
...
</dvt:pivotTable>
```

Figure 26–15 shows the resulting pivot table.

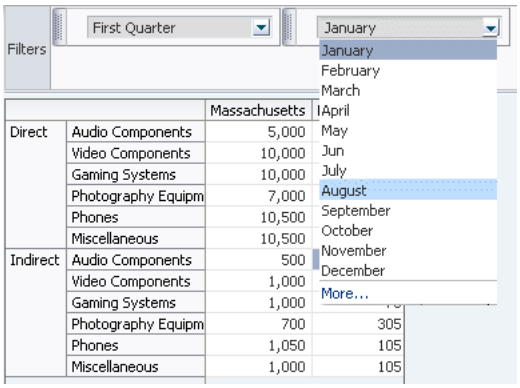
Figure 26–15 Pivot Table with Customized Header Cells

		Sales		Units	
		All Channels		All Channels	
		Go to Tag Guide page		Go to Tag Guide page	
		World	Boston	World	Boston
2007	Tents Go to Tag Guide page	20,000.000	500.000	200.000	50.000
	Canoes Go to Tag Guide page	15,000.000	1,500.000	75.000	8.000
2006	Tents Go to Tag Guide page	10,000.000	250.000	100.000	25.000
	Canoes Go to Tag Guide page	7,500.000	750.000	40.000	4.000
2005	Tents Go to Tag Guide page	5,000.000	125.000	50.000	15.000
	Canoes Go to Tag Guide page	3,750.000	375.000	20.000	2.000

26.11 Using a Pivot Filter Bar with a Pivot Table

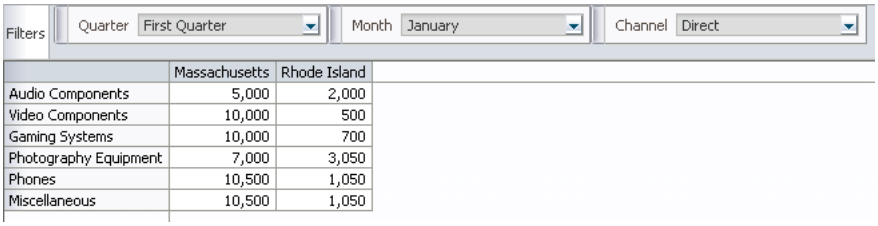
You can enhance the data filtering capacity in a pivot table by adding a pivot filter bar. Zero or more layers of data not already displayed in the pivot table row edge or column edge are displayed in the page edge. Figure 26–16 shows a pivot filter bar with Quarter and Month layers that can be used to filter the data displayed in the pivot table.

Figure 26–16 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 26–17 shows the modified pivot table and pivot filter bar when the Channel data layer is pivoted to the page edge.

Figure 26–17 Pivot Table and Pivot Filter Bar After Pivot



26.11.1 How to Associate a Pivot Filter Bar with a Pivot Table

You associate a pivot filter bar component, `dvt:pivotFilterBar`, to work with a pivot table component, `dvt:pivotTable`, by configuring the data model and associated properties to work with both components. Example 26–10 shows sample code for associating a pivot filter bar with a pivot table.

Example 26–10 Sample Code 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 ADF Pivot Tables" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- Add a pivot filter bar to a pivot table bound to data.

From the ADF Data Visualizations page of the Component Palette, Pivot Table panel, you can drag a `dvt:pivotFilterBar` element above a `dvt:pivotTable` element that has 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.

- Add a pivot filter bar to a pivot table not bound to data.

  From ADF Data Visualizations page of the Component Palette, Pivot Table panel, you can drag a `dvt:pivotFilterBar` element above a `dvt: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.

27

Using ADF Geographic Map Components

This chapter describes how to use an ADF geographic map component with databound themes to display data, and provides the options for map customization.

This chapter includes the following sections:

- Section 27.1, "Introduction toGeographic Maps"
- Section 27.2, "Understanding Data Requirements for Geographic Maps"
- Section 27.3, "Customizing Map Size, Zoom Control, and Selection Area Totals"
- Section 27.4, "Customizing Map Themes"
- Section 27.5, "Adding a Toolbar to a Map"

For information about the data binding of ADF geographic map themes, see the "Creating Databound ADF Geographic Maps" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

27.1 Introduction toGeographic Maps

A geographic map component is a data visualization component that provides the functionality of Oracle Spatial within the framework. This component lets users represent business data on a geographic map and superimpose multiple layers of information (known as *themes*) on a single map.

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 ADF 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 ADF 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.

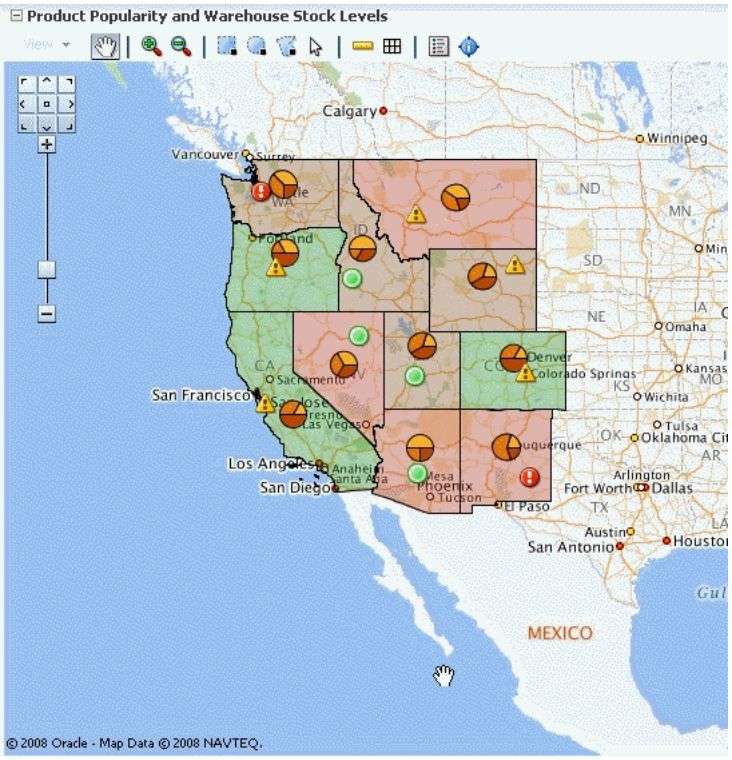
The base map becomes the background on which you build interactive layers of information in JDeveloper using the ADF geographic map component. The ADF geographic map requires that you define at least one layer but you can create as many layers as you wish.

27.1.1 Available Map Themes

The ADF geographic map provides a variety of map themes, each of which must be bound to a data collection. Figure 27–1 shows a map with several themes. 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 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 27–1 Geographic Map of Southwest US with Color, Point, and Pie Graph Themes



27.1.2 Geographic Map Terminology

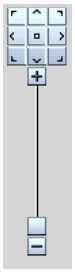
The following list gives a brief description of the terminology used in a geographic map:

- Base map: Provides the background geographic data, zoom levels, and the appearance and presence of items such as countries, cities, and roads. 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: Consists of pan icons and a zoom slider that render in the upper left-hand corner of the map. Figure 27–2 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 `dvt:map` tag. The View menu (which appears in the map toolbar just above the sample map) lets you determine the visibility of the zoom control. By default, the initial zoom level for a map is set to 0.

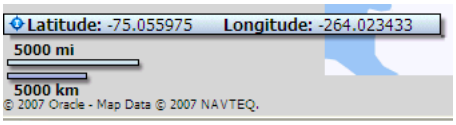
Figure 27–2 Zoom Control of a Map



- Pan icons: Consists of 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: Consists of a 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.

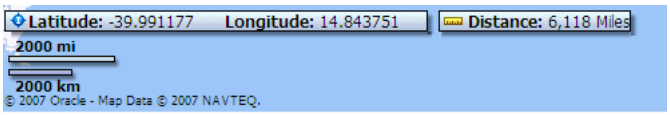
- Scale: Consists of two horizontal bars that display in the lower left-hand corner of the map below the information panel and above the copyright. Figure 27–3 shows the scale. The top bar represents miles (mi) and the bottom bar represents kilometers (km). Labels appear above the miles bar and below 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 27–3 Map Information Panel, Scale, and Copyright



- Information panel: Displays latitude and longitude in the lower left-hand corner above the scale. Figure 27–3 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 27–4 shows the measurement panel with a distance measurement. Area measurement and radius measurement appear in a similar manner with the appropriate labels.

Figure 27–4 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 `dvt:map` tag.
- Overview map: Consists of a miniature view of the main map as shown in Figure 27–5. 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 27–5 Overview Map

The following items are part of the overview map:

- Reticule: Appears as 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: Contains the following elements in the sequence listed:
  - View menu: Lets you control which themes are visible, select a specific theme for display, and determine the visibility of the zoom control, legend, and the information panel.
  - Toolbar buttons: Provide the following tools for interaction with the map: Pan, Zoom In, Zoom Out, Rectangular Selection, Circular Selection, Polygon Selection, Point Selection, Distance, Area, Legend, and Information.

27.1.3 Geographic Map Component Tags

The geometric map has parent tags, map child tags, and tags that modify map themes.

27.1.3.1 Geographic Map Parent Tags

The map component includes the following parent tags:

- **Map tag** `dvt:map`: The parent tag for the main map component. Unlike other data visualization parent tags, the map tag is *not* bound to data. Instead, all the map theme child tags are bound individually to data collections. The map tag 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. For a list and description of the child tags, see Section 27.1.3.2, "Geographic Map Child Tags."
- **Map toolbar** `dvt:mapToolbar`: A parent tag 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 tags.

27.1.3.2 Geographic Map Child Tags

The `dvt:map` tag has the following child tags:

- **Color theme** `dvt:mapColorTheme`: One of the optional map layers that you bind to a data collection.
- **Point theme** `dvt:mapPointTheme`: One of the optional map layers that you bind to a data collection. The point theme identifies individual locations on a map.
- **Bar graph theme** `dvt:mapBarGraphTheme`: One of the optional map layers that you must 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.
- **Pie graph theme** `dvt:mapPieGraphTheme`: One of the optional map layers that you *must* 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.
- **Map legend** `dvt:mapLegend`: Created automatically when you create a map. Use this tag to customize the map legend.
- **Overview map** `dvt: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.

27.1.3.3 Tags for Modifying Map Themes

The following tags modify various map themes:

- **Point style item** `dvt:mapPointStyleItem`: An optional child tag of the `dvt:mapPointTheme` tag. Use this tag only if you want to customize the image that represents points that fall in a certain data value range. To define multiple images, create a tag for each image and specify the associated data value range and image.
- **Pie slice set** `dvt:mapPieSliceSet`: A child of the `dvt:mapPieGraphTheme` tag. This is an optional tag that you use to wrap `dvt:mapPieSliceItem` tags, if you want to customize the color of the slices in a map pie graph.

- **Pie slice item** `dvt:mapPieSliceItem`: A child of the `dvt:mapPieSliceSet` tag. Each pie slice item tag customizes the color of one slice in a map pie graph.
- **Bar graph series set** `dvt:mapBarSeriesSet`: A child of the `dvt:mapBarGraphTheme` tag. This is an optional tag that you use to wrap `dvt:mapBarSeriesItem` tags if you want to customize the color of bars in a map bar graph.
- **Bar graph series item** `dvt:mapBarSeriesItem`: A child of the `dvt:mapBarSeriesSet` tag. Each bar graph series item tag customizes the color of one bar in a map bar graph.

27.2 Understanding Data Requirements for Geographic Maps

The following data requirements apply to the geographic map:

- Configuration requirements include the following information:
  - **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.
  - **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.
- **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.
- **Data controls for map themes:** Each map theme must be bound to a data control. The data control must contain location information that can be bound to similar information in the base map.

27.3 Customizing Map Size, Zoom Control, and Selection Area Totals

You can customize the map size, zoom strategy, appearance of selected regions, and the initial display of the information window and the scale bar.

27.3.1 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.

To adjust the size of a map:

1. In the Structure window, right-click the **dvt:map** node and choose **Go to Properties**.
2. In the **Style** attributes category of the Property Inspector, enter the width and height in the `inlineStyle` attribute.

   For example, to specify a width of 600 pixels and a height of 400 pixels, use the following setting: `inlineStyle="width:600px;height:400px"`.

For a width that uses the entire available width of the page and a height of 400 pixels, use the following setting:
`inlineStyle="width:600px;height:400px".`

27.3.2 How to Specify Strategy for Map Zoom Control

Several attributes on the `dvt:map` tag let you control the initial zoom level, starting location, initial map theme, and zoom strategy.

To control the initial zoom and starting location on a map:

1. In the Structure window, right-click the **dvt:map** node and choose **Go to Properties**.
2. In the **Appearance** attributes category of the Property Inspector, enter values for the following attributes:
   a. In `AutoZoomThemeID`, enter the ID of the first theme that will be displayed.
   b. In `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.
   c. If you want to change the starting location on the map, enter latitude and longitude in `startingX` and `startingY` respectively.
   d. In `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`.

27.3.3 How to Total Map Selection Values

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. The total is displayed in an area under the map. Provide a class that takes `MapSelectionEvent` as an argument in a backing bean method. Example 27–1 shows sample code for a backing bean.

Example 27–1 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 values:

1. In the Structure window, right-click the geographic map node and choose **Go to Properties**.
2. In the Behavior attributes category, for the `SelectionListener` field, enter a method reference that points to the backing bean. For example,

27.4 Customizing Map Themes

You can customize each type of map theme 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.

27.4.1 How to Customize Zoom Levels for a Theme

For all map themes, 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.

To customize the zoom levels of a map theme:

1. In the Structure window, locate the map theme tag (`dvt:mapColorTheme`, `dvt:mapPointTheme`, `dvt:mapBarGraphTheme`, or `dvt:mapPieGraphTheme`) that you want to customize.
2. Right-click the tag and choose **Go to Properties**.
3. In the **Appearance** attributes category of the Property Inspector, enter the desired low and high zoom values for the `MinZoom` and the `MaxZoom` attributes respectively.

27.4.2 How to Customize the Labels of a Map Theme

By default, the `ID` of the map theme is used as the label when that theme is displayed in the legend or in the Theme Selection dialog. However, each map theme tag has the following attributes that serve as optional labels for the theme:

- `shortLabel`: Specifies a label for the theme when displayed in the map legend.
- `menuLabel`: Specifies a label for the theme in the Theme Selection dialog.

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.

To customize the labels of a map theme:

1. In the Structure Window, locate the map theme tag that you want to customize.
2. Right-click the tag node and choose **Go to Properties**.
3. In the **Appearance** attributes category of the Property Inspector, enter text in the `shortLabel` attribute (for display in the legend) and in the `menuLabel` attribute (for display in the 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"
   ```

27.4.3 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.

To customize the colors of a color map theme:

1. In the Structure window, right-click the **dvt:mapColorTheme** node and choose **Go to Properties**.
2. In the **Theme Data** attributes category of the Property Inspector:
   - 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 color 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 `dvt: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.

27.4.4 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.

You define a `dvt:mapPointStyleItem` tag for each custom image that you want to use in a map point theme.

To customize the images for points in a map point theme:

1. In the Structure window, right-click the **dvt:mapPointTheme** node and select **Insert inside dvt:mapPointTheme** > **Point Style Item**.
2. In the ensuing Insert Point Style Item wizard, provide settings for the data value range of this custom point by doing the following:
   a. In Step 1 of the wizard (Common Properties), you must specify the URL for the image that should be displayed on the map for a point that falls in the data value range for this custom image.
   b. In Step 2 of the wizard (Advanced Properties), specify the data value range that this custom image should represent by entering values in MaxValue and MinValue fields.
   c. In the Id field, enter a unique identifier for the custom image that you are defining.
   d. In the Short Label field, 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.
   e. Optionally, specify different image URLs for a hover image and a selected image.
   f. Click **Finish**.
3. Repeat Step 1 and Step 2 for each custom image that you want to create for your point theme.

27.4.5 What Happens When You Customize the Point Images in a Map

When you use the Insert Point Style Item wizard to specify a custom image representing a range of data values for a point theme, a child `dvt:mapPointStyleItem` tag is defined inside the parent `dvt:mapPointTheme` tag. Example 27–2 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 27–2 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>
```

27.4.6 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 by using the `mapBarSeriesSet` tag and the `mapBarSeriesItem` tag.

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.

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 dvt:mapBarGraphTheme** > **Map Bar Series Set**.

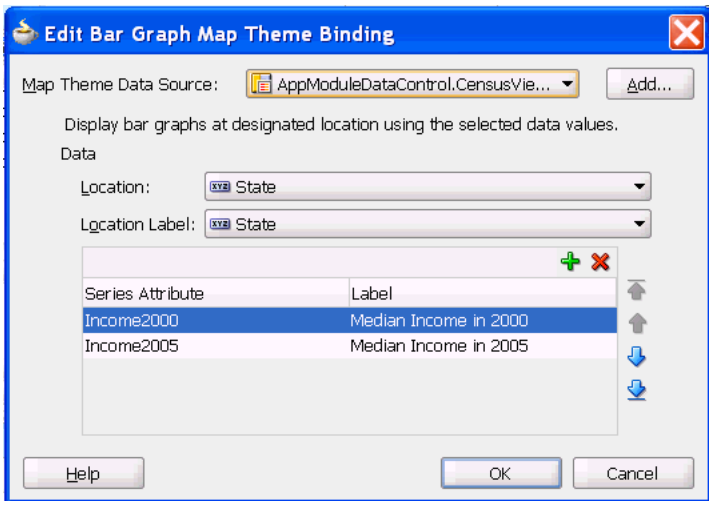
   There are no attributes to set for this tag. It is used to wrap the individual bar series item tags.

2. In the Structure window, right-click the **dvt:mapBarSeriesSet** tag and choose **Insert inside dvt:mapBarSeriesSet** > **Map Bar Series Item**.

3. In the Property Inspector, enter any unique ID and color you want to use for the first bar that appears in the graph.

   To find the sequence of the bars in the graph, examine the Edit Bar Graph Map Theme Binding dialog. The sequence of the entries in the Series Attribute column of that dialog determines the sequence that bars appear in the graph. For example, in Figure 27–6, the first bar in the graph represents Income2000 and the second bar represents Income2005.

Figure 27–6 Edit Bar Graph Map Theme Binding Dialog



4. Repeat Step 3 for every bar in the graph.

27.4.7 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 27–3 shows sample XML code generated when you customize the bars in a map bar graph.

Example 27–3 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>
```

27.4.8 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 by using the `mapPieSlicesSet` tag and the `mapPieSliceItem` tag.

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.

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 dvt:mapPieGraphTheme > Pie Slice Set**.

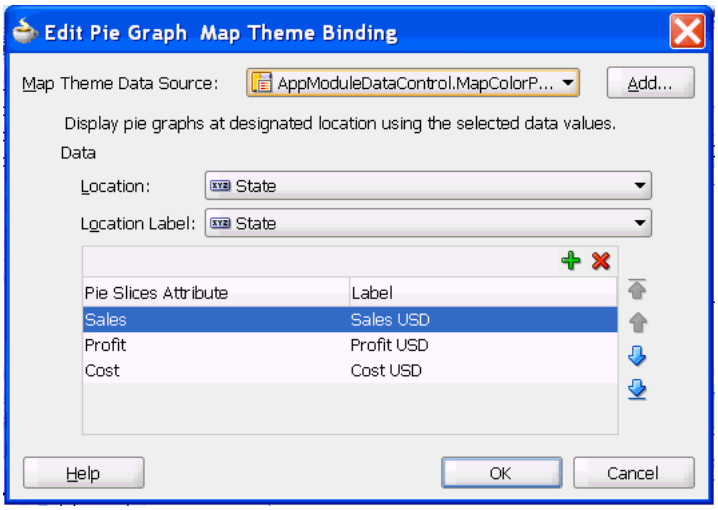
There are no attributes to set for this tag. It is used to wrap the individual pie graph item tags.

2. In the Structure window, right-click the **dvt:mapPieSliceSet** node and choose **Insert inside dvt:mapPieSliceSet > Pie Slice Item**.

3. In the Property Inspector, enter any unique ID and color you want to use for the first slice that appears in the graph.

   To find the sequence of the slices in the graph, examine the Edit Pie Graph Map Theme Binding dialog. The sequence of the entries in the Pie Slices Attribute column of that dialog determines the sequence that slices appear in the graph. For example, in Figure 27–7, the first slice in the graph represents Sales, the second slice represents Profit, and the third slice represents Cost.

Figure 27–7 Edit Pie Graph Map Theme Binding Dialog



4. Repeat Step 3 for every slice in the graph.

27.4.9 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 27–4 shows sample XML code generated in a JSF page when you customize the slices in a map pie graph.

Example 27–4 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>
```

27.5 Adding a Toolbar to a Map

When you create an ADF geographic map, also create a map toolbar if you want to be able to display the legend and the 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.

27.5.1 How to Add a Toolbar to a Map

Because the map toolbar is a component that is separate from the map, you can position the toolbar on the JSF page above or below the map. The following procedure assumes that a map component exists on the JSF page.

To create a map toolbar:

1. In the Structure window, right-click the **dvt:map** node and choose **Insert before dvt:map** or **Insert after dvt:map** > **ADF Data Visualization**.
2. From the ADF Data Visualization Item dialog, select **Toolbar**.
3. In the Insert Toolbar wizard that displays, enter the ID of the map on which this toolbar will operate and click **Next**.
4. Enter a unique ID for the toolbar and optionally change the settings that control the visibility of each tool.

27.5.2 What Happens When You Add a Toolbar to a Map

When you add a toolbar to a map, the following occur:

- A toolbar appears in the JSF page above or below the map as requested. The toolbar contains all the tools unless you change the visibility of one or more tools.
- XML code is generated and appears in the JSF page above or below the code for the map.

Example 27–5 shows sample code for a toolbar that is associated with a map with the ID of `map_us`. It also shows the location of the code for the map.

Example 27–5 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

Using ADF Gantt Chart Components

This chapter describes how to use a databound ADF Gantt chart component to display data, and provides the options for customizing Gantt charts.

This chapter includes the following sections:

For information about the data binding of ADF Gantt charts, see the "Creating Databound ADF Gantt Charts" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

28.1 Introduction to the ADF 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. An ADF 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 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.

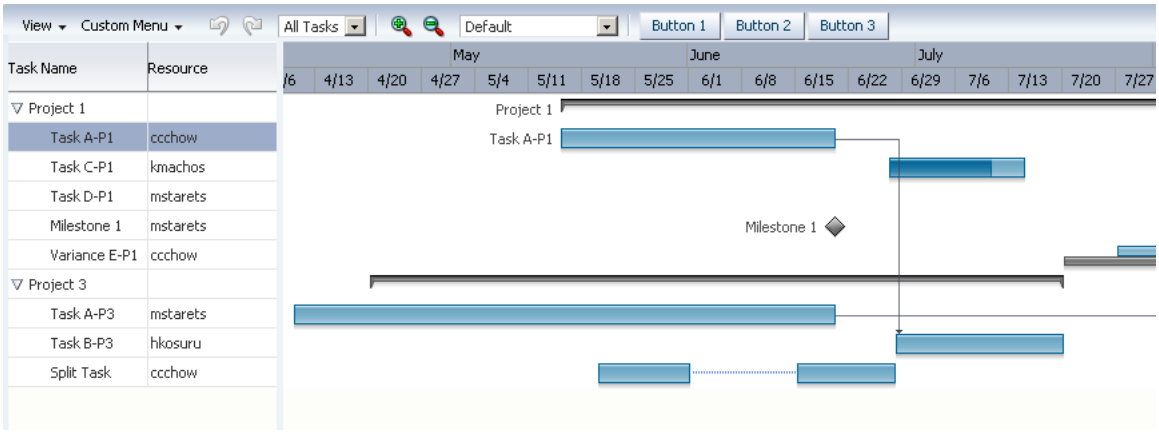
Both Gantt chart regions are based on an ADF Faces tree table component. For more information about ADF tree tables, including virtualization of rows, see Chapter 10, "Using Tables and Trees".

28.1.1 Types of Gantt Charts

The Gantt chart provides the following components:

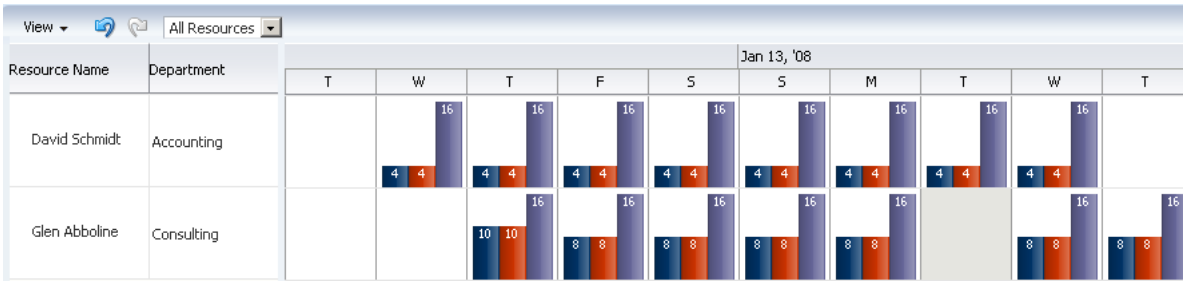
- Project Gantt chart: A project Gantt chart is used for project management. The chart lists tasks vertically and shows the duration of each task as a bar on a horizontal time line. It graphs each task on a separate line as shown in Figure 28–1.

Figure 28–1 Project Gantt Chart for a Software Application



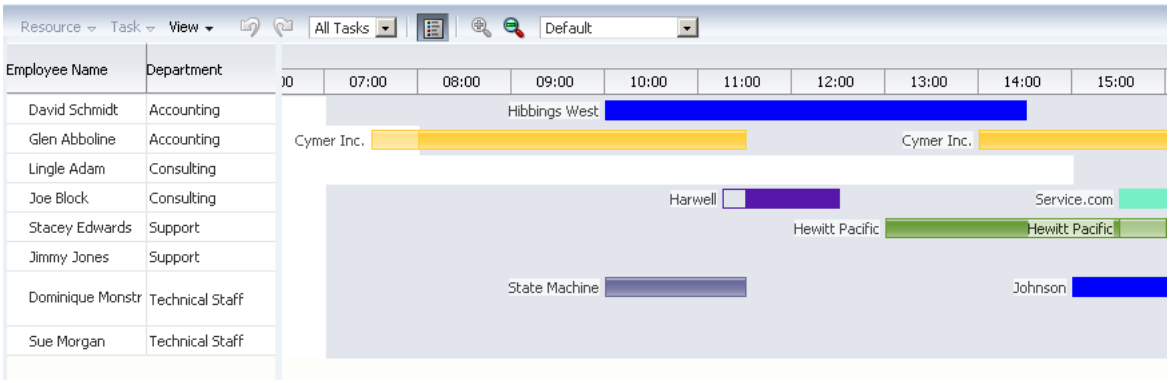
- Resource Utilization Gantt chart: A resource utilization Gantt chart graphically shows the metrics for a resource, for example, whether resources are over or under allocated. It shows resources vertically while showing their metrics, such as allocation and capacity on the horizontal time axis. Figure 28–2 shows a resource utilization Gantt chart illustrating how many hours are allocated and utilized for a particular developer resource in a given time period.

Figure 28–2 Resource Utilization Gantt Chart for a Software Application



- Scheduling Gantt chart: A scheduling Gantt chart is used for resource scheduling. The chart is based on manual scheduling boards and shows resources vertically, with corresponding activities on the horizontal time axis. Examples of resources include people, machines, or rooms. The scheduling Gantt chart uses a single line to graph all the tasks that are assigned to a resource as shown in Figure 28–3.

Figure 28–3 Scheduling Gantt Chart for a Software Application



28.1.2 Functional Areas of a Gantt Chart

A Gantt chart consists of the following functional areas:

- Table region: 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 in the Gantt chart data binding dialogs.

  For example, in Figure 28–1, the table region contains the following columns: Name (of the task), Priority, Orig. Est., Curr. Est., Elapsed (days), Remaining (days), and Resources.

- Chart region: Displays a bar graph of the Gantt chart data along a horizontal time axis. The time axis provides for major and minor settings to allow for zooming. The major setting is for larger time increments and the minor setting is for smaller time increments.

  For example, in Figure 28–1, the chart region graphs tasks on a time axis that shows days within weeks.

- Information panel: Displays both the information region that displays text about the selected task or metrics about the selected resource, and the optional legend that displays task types in the area beneath the table region and the chart region. Note that the Gantt chart legend is not present unless you insert the legend child tag inside the parent Gantt chart tag.

- Toolbar: Lets users perform operations on the Gantt chart. The toolbar is visible in the Gantt chart by default. You can change the visibility of the toolbar by setting the `ShowToolbar` attribute on the Appearance page of the Property Inspector for the Gantt chart.

  The toolbar consists of the following sections:

  – Menu bar: The left section of the toolbar contains a set of menus for the Gantt chart. Each Gantt chart type has a set of default options. Figure 28–4 displays the menu bar, which is visible in the Gantt chart by default. You can change the visibility of the menu bar by setting the `ShowMenuBar` attribute in the Appearance page of the Property Inspector for the Gantt chart. You can customize menu items by using the `menubar` facet.

  Note: The 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.

Figure 28–4 Sample Menu Bar for a Gantt Chart

– Toolbar buttons: The right section of the toolbar displays a set of action buttons for working with the Gantt chart. Each Gantt chart type has a set of default options. Figure 28–5 shows a sample toolbar for a project Gantt chart.

Figure 28–5 Sample Toolbar for a Project Gantt Chart

You can customize toolbar buttons by using the `toolbar` facet.

- Context menu: Right-clicking in the Gantt chart table or chart regions provides a popup 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.
- Printing service: The Gantt chart provides printing capability in conjunction with XML Publisher by generating PDF files. For more information, see Section 28.9, "Printing a Gantt Chart".

28.1.3 Description of Gantt Chart Tasks

Project and scheduling Gantt charts use predefined tasks with a set of formatting properties that describe how the tasks will be rendered in the chart area. All supported tasks must have a unique identifier. The following describes the supported tasks and how they appear in a Gantt chart:

- Normal: The basic task type. It is a plain horizontal bar that shows the start time, end time, and duration of the task.
- Summary: The start and end date for a group of subtasks. A summary task cannot be moved or extended. Instead, it is the responsibility of the application to execute code to recalculate the start and end date for a summary task when the date of a subtask changes. Summary tasks are available only for the project Gantt chart.
- Milestone: A specific date in the Gantt chart. There is only one date associated with a milestone task. A milestone task cannot be extended but it can be moved. A milestone task is available only for the project Gantt chart.
- Recurring: A task that is repeated in a Gantt chart, each instance with its own start and end date. Individual recurring tasks can optionally contain a subtype. All other properties of the individual recurring tasks come from the task which they are part of. However, if an individual recurring task has a subtype, this subtype overrides the task type.
- Split: A task that is split into two horizontal bars, usually linked by a line. The time between the bars represents idle time due to traveling or down time.
- Scheduled: The basic task type for a scheduling Gantt chart. This task type shows the starting time, ending time, and duration of a task, as well as startup time if one is specified.

For normal, summary, and milestone tasks, additional attributes are supported that would change the appearance and activity of a task. These style attributes include:

- `percentComplete`, `completedThrough`: An extra bar would be drawn to indicate how far the task is completed. This is applicable to normal and summary task types.
- `critical`: The color of the bar would be changed to red to mark it as critical. This is applicable to normal, summary, and milestone task types.
- `actualStart` and `actualEnd`: When these attributes are specified, instead of drawing one bar, two bars are drawn. One bar indicates the base start and end date, the other bar indicates the actual start and end date. This is applicable to normal and milestone task types.

Figure 28–6 displays a legend that shows common task types in a project Gantt chart.

Figure 28–6 Project Gantt Chart Legend for Task Types

28.2 Understanding Gantt Chart Tags and Facets

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:

- `dvt:projectGantt`
- `dvt:resourceUtilizationGantt`
- `dvt:schedulingGantt`

All Gantt chart components support the child tag `dvt: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 28–1 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 28–1 Sample Code for Project Gantt Chart Columns

```
<dvt:projectGantt startTime="2008-04-12"
           endTime="2009-04-12"
           value="#{project.model}"
           var="task">
  <f:facet name="major">
    <dvt:timeAxis scale="months"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="weeks"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column headerText="Task Name">
      <af:outputText value="#{task.taskName}"/>
    </af:column>
```

```
  </f:facet>
    <af:column headerText="Resource">
      <af:outputText value="#{task.resourceName}"/>
    </af:column>
    <af:column headerText="Start Date">
      <af:outputText value="#{task.startTime}"/>
    </af:column>
    <af:column headerText="End Date">
      <af:outputText value="#{task.endTime}"/>
    </af:column>
</dvt:projectGantt>
```

In addition to the `nodeStamp` facet, other facets are used for customizations by the Gantt chart components. Table 28–1 shows the facets supported by Gantt chart components.

Table 28–1 Facets Supported by Gantt Chart Components

Name	Description
`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:toobar` component.

For complete descriptions of all the Gantt chart tags, their attributes, and a list of valid values, consult the DVT tag documentation. To access this documentation for a specific tag in JDeveloper, select the tag in the Structure window and press **F1**. To access the full ADF Data Visualization Tools tag library in JDeveloper Help, expand the Javadoc and Tag Library References node in the online Help Table of Contents and click the link to the tag library in the JDeveloper Tag Library Reference topic.

28.3 Understanding Gantt Chart User Interactivity

At runtime, users can perform a wide range of operations on a Gantt chart, including navigation and display, as well as actions that change the data in the table or chart region.

When a user interaction involves a change in data, the Gantt chart processes the change by performing validation, event handling, and update of the data model. 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.

28.3.1 Navigating in a Gantt Chart

You can browse through Gantt chart regions by scrolling, or you can access a specific date in the chart region. You can also control if columns in the table region are visible.

28.3.1.1 Scrolling and Panning the List Region or the Chart Region

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.

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.

In project and scheduling Gantt charts, users can pan the chart area by dragging it vertically and horizontally using the mouse. A move cursor displays when the user clicks inside the chart area, other than on a task.

28.3.1.2 How to Navigate to a Specific Date in a Gantt Chart

You can move the chart region of the Gantt chart rapidly to a specific date.

To navigate to a specific date in a Gantt chart:

1. From the View menu, choose **Go to Date**.
2. In the Go to Date dialog, specify the desired date by clicking the **Select Date** icon and indicating the date in the calendar.

3. Click **OK**.

The display of the chart region of the Gantt chart begins at the date you requested.

28.3.1.3 How to Control the Visibility of Columns in the Table Region

By default, all the columns that you define when you create a databound Gantt chart are visible in the table region. You can selectively cause one or more of these columns to be hidden.

To control the display of columns in the table region of a Gantt chart:

1. From the View menu, select **List Pane**.
2. From the context menu, select **Columns**.
3. In the Columns menu, deselect any column that you want to be hidden in the table region of the Gantt chart. You can also select any column that you want to make visible in the table region.

> **Note:** You must keep at least one column visible in the table region.

28.3.2 How to Display Data in a Hierarchical List or a Flat List

If a Gantt chart is using a hierarchical data model, then you have the option of displaying all the Gantt chart data in a collapsed form or in an expanded form.

To control the display of Gantt chart data in a list:

1. From the View menu, select **List Pane**.
2. From the ensuing menu, select either **Show As List**, for an expanded list, or **Show As Hierarchy**, for a collapsed list.

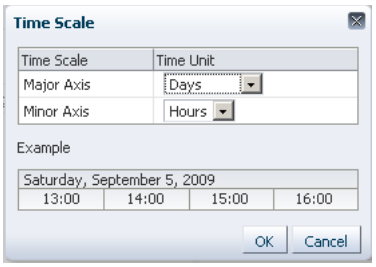
28.3.3 How to Change the Gantt Chart Time Scale

You can change the time scale display in a Gantt chart and 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.

To change the settings of a time axis:

1. From the View menu, select **Time Scale**.
2. In the ensuing Time Scale dialog, in the Time Unit column, select a new unit value for either the major axis, the minor axis, or both axes. A sample box displays sample settings for the time unit that you select. Figure 28–7 shows the Time Scale dialog.

Figure 28–7 Time Scale Dialog



3. Click **OK**.

To zoom in or out on a time axis:

1. Optionally, on the toolbar, click the **Zoom In** icon to display the time axis at a lower level time unit.
2. Optionally, on the toolbar, click the **Zoom Out** icon to display the time axis at a higher level time unit.
3. Optionally, in the box on the toolbar after the zoom icons, select a time period that represents the amount of time on the chart that you want to display without the need to scroll.
4. Optionally, right-click the time axis for which you wish to change the scale and select an available time unit from the submenu.

28.4 Understanding Data Requirements for the Gantt Chart

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. For more information, see the "Creating Databound ADF Gantt Charts" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

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.

28.4.1 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 28–2 shows a complete list of data object keys for the project Gantt chart.

Table 28–2 Data Object Keys for Project Gantt Chart

Data Object Key	Data Type and Description
`actualEnd`	Date. The actual end time for normal and milestone tasks.
`actualStart`	Date. The actual start time for normal and milestone tasks.
`completedThrough`	Date. Completed through for normal and summary tasks.
`critical`	Boolean. Specifies whether or not the task is critical for all tasks.
Dependency (node)	A list of dependencies for a task. Data object keys for dependencies include: ■ `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 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.

Table 28–2 (Cont.) Data Object Keys for Project Gantt Chart

Data Object Key	Data Type and Description
`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.

28.4.2 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.

Table 28–3 shows a complete list of data object keys for the resource utilization Gantt chart.

Table 28–3 Data Object Keys for the Resource Utilization Gantt Chart

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.
`time` (required)	Date. The start time of the time bucket.
`values` (required)	Double. The values of the metrics.

28.4.3 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 28–4 shows a complete list of data object keys for a scheduling Gantt chart.

Table 28–4 Data Object Keys for Scheduling Gantt Charts

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 the 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 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`.
`isContainer`	Boolean. Specifies whether or not a node definition is a container.
`label`	String. The label associated with the task bar for all tasks.

Table 28–4 (Cont.) Data Object Keys for Scheduling Gantt Charts

Data Object Key	Data Type and Description
`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`.
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.

28.5 Creating an ADF Gantt Chart

You can use any of the following data sources to create an ADF Faces Gantt chart component:

- ADF Data Controls: You declaratively create a databound Gantt chart by dragging and dropping a data collection from the ADF Data Controls panel. You can create a Gantt chart using a data collection that provides row set data as described in the "Creating Databound ADF Gantt Charts" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
- Data Model: You programmatically specify the data model for the Gantt chart by providing an EL expression that references a backing bean method using the `value` attribute of the Gantt tag.

28.6 Customizing Gantt Chart Legends, Toolbars, and Context Menus

You can modify default Gantt chart features including the information panel and legend that are displayed below the Gantt chart, menu bar options and toolbar buttons, and the popup menu that is displayed when you right-click in the Gantt chart table or chart regions.

28.6.1 How to Customize 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 `dvt: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 28–2 show sample code to display information about an On Hold task in the legend of a project Gantt chart.

Example 28–2 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>
```

28.6.2 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 28–3 shows sample code for specifying a new menu item.

Example 28–3 Sample Code for Custom Menu Item

```
<dvt:projectGantt var="task">
<f:facet name=”menus”>
  <af:menu text=”My Menu”>
    <af:commandMenuItem text="Add..." />
    <af:commandMenuItem text="Create.." />
  </af:menu>
</f:facet>
</dvt:projectGantt>
```

Example 28–4 shows sample code for specifying a new toolbar button.

Example 28–4 Sample Code for Custom Toolbar Button

```
<dvt:schedulingGantt var="task">
<f:facet name="toolbar">
  <af:toolbar>
    <af:commandToolbarButton 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 28–5 shows sample code in a backing bean.

Example 28–5 Backing Bean for Handling Data Change

```
public void handleDataChanged(DataChangeEvent evt)
{
if (DataChangeEvent.DELETE == evt.getActionType())
            ..........
}
```

28.6.3 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 28–6 shows sample code for specifying a custom menu item in the table region context menu.

Example 28–6 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 28–7 shows sample code to update a custom popup menu on a task bar based on which task is selected in the chart region of a project Gantt chart.

Example 28–7 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 28–8 shows sample code for the backing bean.

Example 28–8 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 13, "Using Popup Dialogs, Menus, and Windows".

28.7 Working with Gantt Chart Tasks and Resources

You can customize Gantt chart tasks to create a new task type, specify a custom data filter, and add a double-click event to a task bar.

28.7.1 How to Create 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 `.jspx` file 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 28–9 shows sample code to define the properties for a custom task type in the `.jspx` file.

Example 28–9 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>
```

shows sample code to define a `TaskbarFormat` object fill and border color and register the object with the `taskbarFormatManager`.

Example 28–10 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);
```

28.7.2 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 28–11 shows sample code for registering a Resource filter object with the `FilterManager`.

Example 28–11 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 28–12 shows sample code for a `FilterListener`.

Example 28–12 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
        }
    }
```

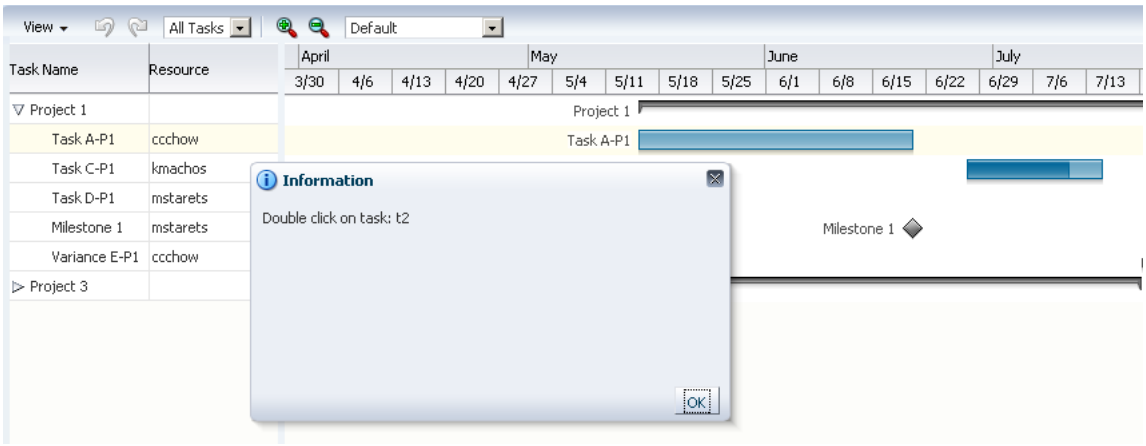
To specify a custom data filter:

1. In the Structure window, right-click the Gantt chart node and choose **Go to Properties**.
2. In the Behavior category of the Property Inspector, in the `FilterListener` field, enter a method reference to the `FilterListener` you defined. For example, `"#{project.handleFilter}"`.

28.7.3 How to Add a Double-Click Event to a Task Bar

Gantt chart components support a double-click event on a task bar. For example, you may want to display detailed information about a task in a popup window. Figure 28–8 shows a project Gantt chart with a double-click event on a task bar.

Figure 28–8 Task Bar with Double-Click Event



Example 28–13 show sample code for adding a double-click event to a task bar.

Example 28–13 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)
```

28.8 Specifying Nonworking Days, Read-Only Features, and Time Axes

You can customize a Gantt chart to display nonworking days of the week, turn off user interaction features, and specify the time axes.

28.8.1 Identifying 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.

28.8.1.1 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.

To identify weekdays as nonworking days:

1. In the Structure window, right-click a Gantt chart node and choose **Go to Properties.**
2. In the Appearance category of the Property Inspector, 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 28–14 shows sample code in a backing bean.

Example 28–14 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.

28.8.1.2 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.

To identify specific dates as nonworking days:

1. In the Structure Window, right-click a Gantt chart and choose **Go to Properties.**
2. In the Property Inspector, 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.

28.8.2 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 28–5 shows the valid values and the disabled feature for the Gantt chart types.

Table 28–5 Valid Values for Read-Only Attributes

Value	Feature Disabled
`clipboard`	Cut, copy, and paste tasks for all Gantt charts.
`edit`	Changes to the data model for all Gantt charts.
`indenting`	Indent and outdent tasks for project and scheduling Gantts charts.
`legend`	Hide and show legend and task information for all Gantt charts.
`linking`	Link and unlink tasks for scheduling Gantt charts.
`print`	Print task for all Gantt charts.
`properties`	Show property dialogs for all Gantt charts.
`split`	Split task for project Gantt.
`undo`	Undo and redo tasks for all Gantt charts.
`view`	Show as list, Show as hierarchy, Columns, Expand and Collapse tasks for all Gantt charts.
`zoom`	Changes to the zoom level for all Gantt charts.

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 Property Inspector, 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}"`.

28.8.3 Customizing 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 28–18 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 28–15 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.

28.8.3.1 How to Create and Use a Custom Time Axis

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.

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 28–16 show sample code for the interface.

Example 28–16 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 Property Inspector, 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 Property Inspector, set the **Scale** attribute for major and minor time axis, and specify the **ZoomOrder** attribute to zoom to the custom times scales. Example 28–17 shows sample code for setting a `threeyears` minor time axis and a `fiveyears` major time axis.

Example 28–17 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>
```

28.9 Printing a Gantt Chart

The ADF Gantt chart provides a helper class (`GanttPrinter`) that can generate a Formatted Object (FO) for use with XML Publisher to produce PDF files.

28.9.1 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 28–18 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 28–18 Code for Setting the Time Period Option for Printing a Gantt Chart

```
_printer.setStartTime(startDate);
_printer.setEndTime(endDate);
```

28.9.2 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 28–19 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 28–19 Sample ActionListener for Handling the Gantt Chart Print Event

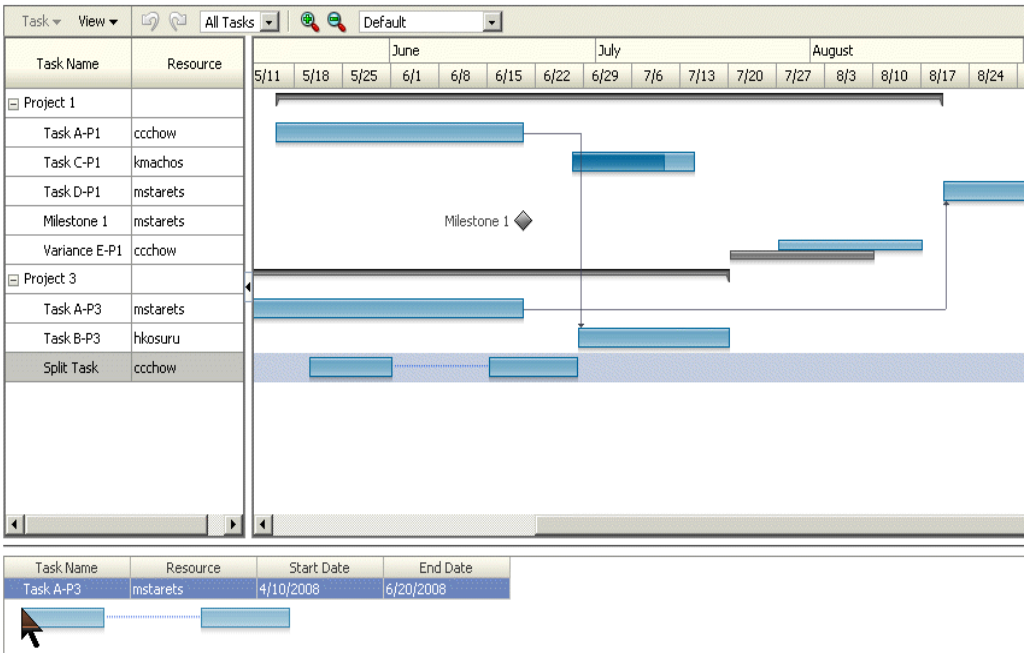
```
public void handleAction(GanttActionEvent evt)
{
  if (GanttActionEvent.PRINT == evt.getActionType())
  {
     FacesContext _context = FacesContext.getCurrentInstance();
     ServletResponse _response = (ServletResponse)
              _context.getExternalContext().getResponse();
     _response.setContentType("application/pdf");
     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();
   }
}
```

28.10 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. shows a project Gantt chart with tasks dragged from the table region onto a table of tasks.

Figure 28–9 Project Gantt Chart as Drag Source



Example 28–20 shows sample code for adding drag and drop functionality to a scheduling Gantt chart.

Example 28–20 Sample Code for Adding Drag and Drop Functionality

```
<dvt:schedulingGantt value="#{test.treeModel}"
    ..............
    <af:schedulingDragSource actions="COPY MOVE" modelName="treeModel"  />
</dvt:projectGantt>
```

Example 28–21 shows sample code for the listener method for handling the drop event.

Example 28–21 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 32.4, "Adding Drag and Drop Functionality for Collections".

29

Using ADF Hierarchy Viewer Components

This chapter describes how to use an ADF hierarchy viewer component to display data, and provides the options for hierarchy view customization.

This chapter includes the following sections:

- Section 29.1, "Introduction to Hierarchy Viewers"
- Section 29.2, "Data Requirements for Hierarchy Viewers"
- Section 29.3, "Managing Nodes in a Hierarchy Viewer"
- Section 29.4, "Navigating in a Hierarchy Viewer"
- Section 29.5, "Adding Interactivity to a Hierarchy Viewer Component"
- Section 29.6, "Using Panel Cards"
- Section 29.7, "Customizing the Appearance of a Hierarchy Viewer"
- Section 29.8, "Adding Search to a Hierarchy Viewer"

For information about the data binding of ADF hierarchy viewers, see the "Creating Databound Hierarchy Viewers" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

29.1 Introduction to Hierarchy Viewers

A hierarchy viewer component can be used to visually display hierarchical data. Hierarchical data contains master-detail relationships within the data. For example, you could create a hierarchy viewer component that renders an organization chart from a data collection that contains information about the relationships between employees in an organization.

29.1.1 Understanding the Hierarchy Viewer Component

JDeveloper generates the following elements in JSF pages when you drag and drop components from the Component Gallery 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 Viewers" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

- `dvt:hierarchyViewer`

  This element wraps the `dvt:node` and the `dvt:link` elements.

- `dvt: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 `dvt:node` element to reference whatever data you want to display. The `dvt: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 `dvt: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."

- `dvt:link`

  You set values for the attributes of the `dvt: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 29.7.4, "How to Configure the Display of Links and Labels."

- `dvt:panelCard`

  The panel card element provides a method to dynamically switch 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 29.6, "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 Component Gallery to create a hierarchy viewer. For more information see, Section 29.1.3, "Available Hierarchy Viewer Layout Options."

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 Component Palette. Code related to the hierarchy viewer elements is highlighted in the example.

Example 29–1 Hierarchy Viewer Sample Code

```
<dvt:hierarchyViewer id="hierarchyViewer1" layout="hier_vert_top"
                     inlineStyle="width:100%;height:600px;">
  <dvt:link linkType="orthogonalRounded" id="l1"/>
    <dvt:node width="233" height="330" id="n1">
      <f:facet name="zoom100">
        <af:panelGroupLayout layout="vertical"
                             inlineStyle="width:100%;height:100%;padding:5px"
                             id="pgl2">
         <af:panelGroupLayout layout="horizontal" id="pgl3">
         <af:panelGroupLayout inlineStyle="width:85px;height:120px"
```

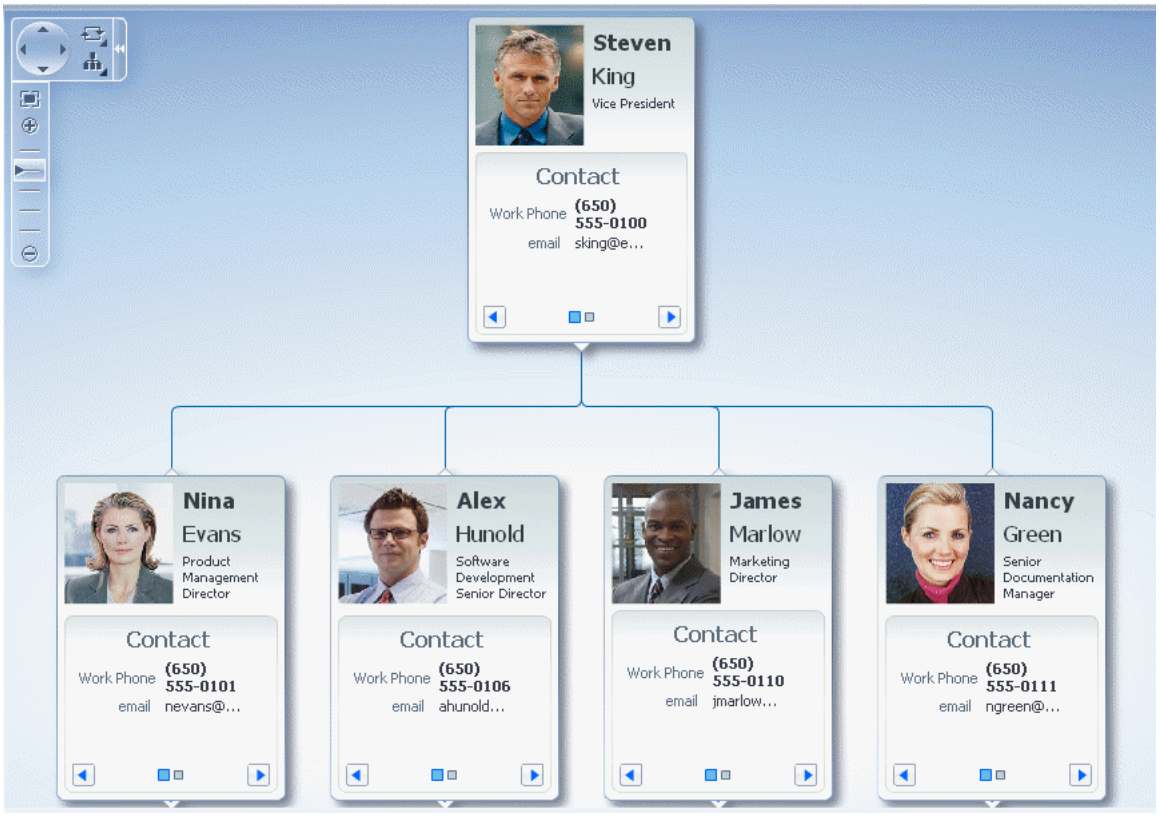
```
                    id="pgl4">
   <af:image id="i1"/>
</af:panelGroupLayout>
<af:spacer width="5" height="5" id="s2"/>
<af:panelGroupLayout layout="vertical" id="pgl1">
  <af:outputText value=" attribute value1 "
              inlineStyle="font-weight:bold;font-size:20px;color:#383A47"
              id="ot5"/>
  <af:outputText value=" attribute value2"
              inlineStyle="font-size:20px;color:#383A47"
              id="ot9"/>
  <af:outputText value=" attribute value3"
              inlineStyle="font-size:11px;font-style:italic;color:#383A47"
              id="ot2"/>
</af:panelGroupLayout>
</af:panelGroupLayout>
<af:spacer height="5" id="s1"/>
<af:separator id="s4"/>
<af:spacer height="5" id="s3"/>
  <dvt:panelCard effect="slide_horz"
              inlineStyle="border-width:1px;border-color:#CCCCCC;
              font-size:16px;font-weight:bold;color:#5a6a7a"
              id="pc1">
    <af:showDetailItem text="first group title "
              inlineStyle="font-weight:bold;font-size:16px;color:#5a6a7a"
              id="sdi1">
    <af:panelFormLayout inlineStyle="width:100%;height:100%;padding:5px"
              id="pfl1">
    <af:panelLabelAndMessage label="attribute label4"
               labelStyle="font-size:14px;color:#5A6A7A" id="plam2">
    <af:outputText value="attribute value4"
                inlineStyle="font-size:14px;color:#383A47" id="ot3">
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="attribute label5"
                labelStyle="font-size:14px;color:#5A6A7A" id="plam6">
    <af:outputText value="attribute value5"
                inlineStyle="font-size:14px;color:#383A47" id="ot8">
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="attribute label6"
                labelStyle="font-size:14px;color:#5A6A7A" id="plam5">
    <af:outputText value="attribute value6"
                inlineStyle="font-size:14px;color:#383A47" id="ot6">
    </af:panelLabelAndMessage>
    </af:panelFormLayout>
  </af:showDetailItem>
  <af:showDetailItem text="second group title "
                inlineStyle="font-weight:bold;
                font-size:16px;color:#5a6a7a" id=sd12">
    <af:panelFormLayout inlineStyle="width:100%;height:100%;padding:5px"
                id="pfl2">
    <af:panelLabelAndMessage label="attribute label7"
                labelStyle="font-size:14px;color:#5A6A7A" id="plam3">
    <af:outputText value="attribute value7"
                inlineStyle="font-size:14px;color:#383A47" id="ot7"/>
    </af:panelLabelAndMessage>
    <af:panelLabelAndMessage label="attribute label8"
                labelStyle="font-size:14px;color:#5A6A7A"id="plam4">
    <af:outputText value="attribute value8"
                inlineStyle="font-size:14px;color:#383A47" id="ot1"/>
    </af:panelLabelAndMessage>
```

```
            <af:panelLabelAndMessage label="attribute label9"
                          labelStyle="font-size:14px;color:#5A6A7A" id="plam1">
            <af:outputText value="attribute value9"
                           inlineStyle="font-size:14px;color:#383A47" id="ot4"/>
            </af:panelLabelAndMessage>
            </af:panelFormLayout>
            </af:showDetailItem>
          </dvt:panelCard>
        </af:panelGroupLayout>
      </f:facet>
    </dvt:node>
</dvt:hierarchyViewer>
```

29.1.2 Hierarchy Viewer Elements and Terminology

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.

Figure 29–1 Hierarchy Viewer Component with Control Panel and Nodes



The Control Panel provides controls so that a user can 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 component, as illustrated by Figure 29–2.

Figure 29–2 Control Panel in Hidden State

You cannot configure the Control Panel to appear in another location. Users click the **Hide** or **Show Control Panel** button shown in Figure 29–2 to hide or expand the Control Panel. Figure 29–3 shows the expanded Control Panel.

Figure 29–3 Control Panel in Show State

You can configure the hierarchy viewer component so that the Control Panel does not appear to the user at runtime. For information about the procedure, see Section 29.7.3, "How to Configure the Display of the Control Panel."

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 `dvt:panelCard` element with one or more `af:showDetailItem` elements. 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

For more information about layouts for a hierarchy viewer component, see Section 29.1.3, "Available Hierarchy Viewer Layout Options."

Table 29–1 Elements in the Control Panel

Control	Name	Description
	Pan Control	Allows user to reposition the hierarchy viewer component within the viewport.

Table 29–1 (Cont.) Elements in the Control Panel

Control	Name	Description
	Zoom to Fit	Allows user to zoom a hierarchy viewer component so that all nodes are visible within the viewport.
	Zoom Control	Allows user to zoom the hierarchy viewer component.
	Hide or Show	Hides or shows the Control Panel.
	Panel Selector	Displays a list of node panels 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 one of the layout options presented by the component. For more information, see Section 29.1.3, "Available Hierarchy Viewer Layout Options."

Hierarchy viewers support state management for 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 `lateralNavigationRowKeys`.

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 support bi-directional support for flipping panel cards from one node view to the next.

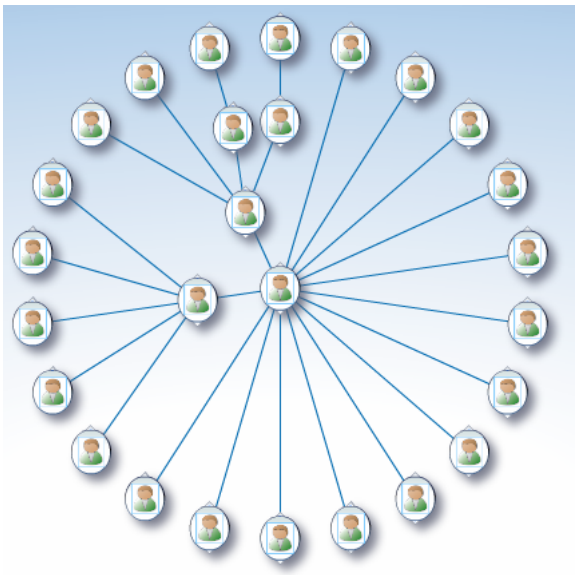
29.1.3 Available Hierarchy Viewer Layout Options

The hierarchy viewer can use any of the following layouts, specified by the component's `type` attribute:

- `hier_vert_top` - Vertical top down
- `hier_vert_bottom` - Vertical bottom up
- `hier_horz_left` - Horizontal left to right
- `hier_horz_right` - Horizontal right to left
- `hier_horz_start` - Horizontal, direction depends on the locale
- `hier_horz_end` - Horizontal, direction depends on the locale
- `tree` - Tree, indented tree
- `radial` - Radial, root node in center and successive child levels radiating outward from their parent nodes
- `circle` - Circle, root node in center and all leaf nodes arranged in concentric circle, with parent nodes arranged within the circle

  Figure 29–4 shows an example of a circle layout for a hierarchy viewer component.

Figure 29–4 Hierarchy Viewer Circle Layout

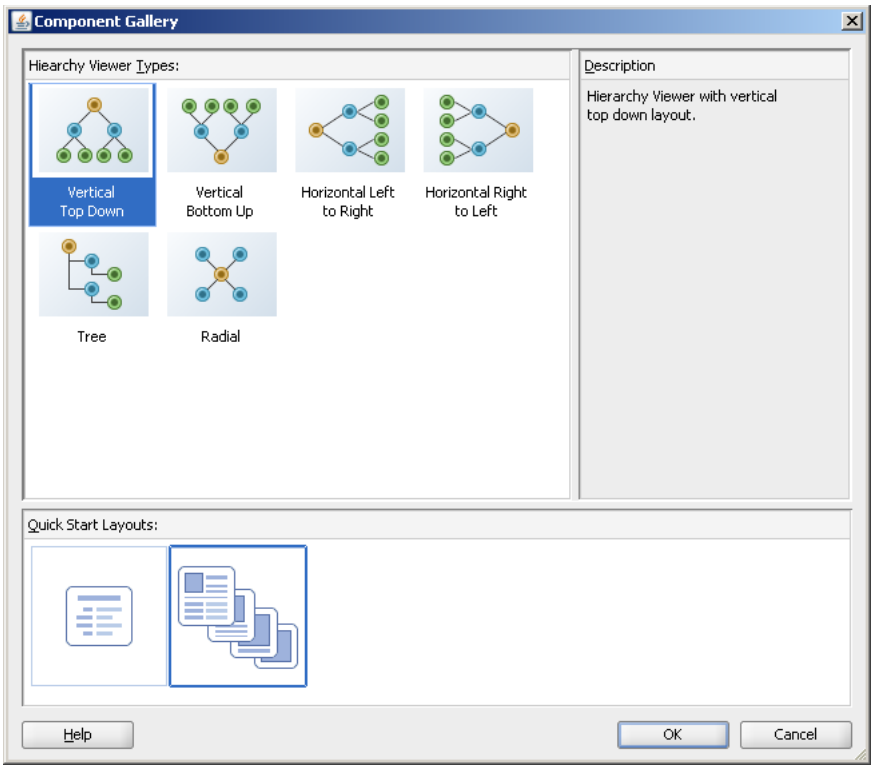


You can define the initial layout of the hierarchy viewer when you insert the component on the page from either the Data Controls panel to bind a data collection to the hierarchy viewer component, or from the Component Palette to insert the component and bind to data later. The available layouts are displayed in the Hierarchy Viewer Types area of the Component Gallery, shown in Figure 29–5.

> **Note:** The circle layout is not available in the Component Gallery. In order to create a hierarchy viewer with a circle layout, you must specify a `circle` value in the `dvt:hierarchyViewer` tag `type` attribute in the Property Inspector.

In the Quick Start Layouts area of the Component Gallery you can also choose to generate the `dvt:panelCard` element to support multiple sets of content for a node, the selection shown in Figure 29–5.

Figure 29–5 Component Gallery for Hierarchy Viewer Components



29.1.4 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.

29.2 Data Requirements for Hierarchy Viewers

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 the *Oracle Fusion Middleware Fusion Developer's Guide for 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 29.4, "Navigating in a Hierarchy Viewer."

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 allows users to do the following:

- Navigate to other nodes in a hierarchy viewer component

  The top of each node contains a single Restore or Isolate button to either display the parent node or single out the node as the anchor node in the hierarchy viewer. One exception is the node at the very top of the hierarchy viewer component, because this node has no parent nodes and may not be isolated.

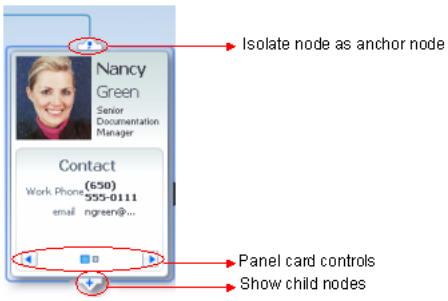
- Show or hide child nodes of the currently selected node in a hierarchy viewer component

  The single **Show** or **Hide** button appears on the bottom of every node that is a not a leaf node. 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 in the same way as an `af:tree` component. For more information, see Section 10.5.4, "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 29.6, "Using Panel Cards."

Figure 29–6 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 29–6 Node Controls



There are four basic types of nodes:

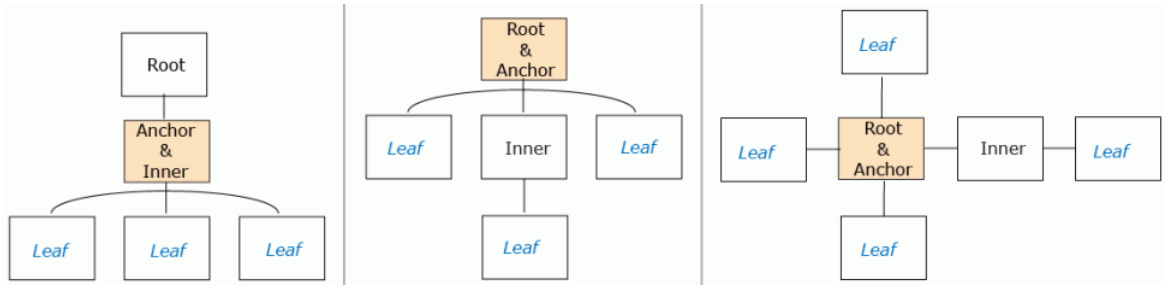
- *Root* nodes are the uppermost visible nodes in a hierarchy viewer component. A root node is always the root of the hierarchy returned from the tree component. Typically, only one root node is visible at a time. However, there could be more than one root node depending on the use case that you implement (for example, in a family tree).
- An *anchor* node is the node that has focus whenever the hierarchy viewer component is rendered. There is always just one anchor node visible.

  The anchor node may be the same as the root node if child nodes are defined for the tree node and if the value of the hierarchy viewer component's `displayLevelsAncestor` property is equal to `0`. At runtime, if a user double-clicks another node that has a value specified for its `setAnchorListener` property, that node becomes the anchor node. An anchor node may also be an inner or leaf node, depending on whether or not it has child nodes. For information about how to specify an anchor node, see Section 29.3.4, "How to Associate a Node Definition with a Particular Set of Data Rows."

  You can specify one or more ancestor levels above the anchor node. For more information, see Section 29.3.5, "How to Specify Ancestor Levels for an Anchor Node."
- *Inner* nodes are nodes that have child nodes.
- *Leaf* nodes are nodes that do not have child nodes.

Figure 29–7 illustrates how a node can be a different type depending on the layout of the hierarchy viewer component.

Figure 29–7 Node Types and Positions



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 Viewers" section in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

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 references 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 the department's name and number.

At runtime, when a user moves the mouse over a node at any zoom level, 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 when the node has been selected in the hierarchy viewer.

Each of the facet elements that you insert can be used to reference other components. You can use one or more of the following components when you define content for a node in a hierarchy viewer component. The node component facet's support the following components:

- `af:commandButton`
- `af:commandImageLink`
- `af:commandLink`
- `af:commandMenuItem`
- `af:goButton`
- `af:goLink`
- `af:image`

  For information about how to use the `af:image` component, see Section 29.7.2, "How to Include Images in a Hierarchy Viewer."
- `af:menu`
- `af:outputFormatted`
- `af:outputText`
- `af:panelFormLayout`
- `af:panelGroupLayout`

  For information about how to use the `panelGroupLayout` component, see Section 8.12.1, "How to Use the panelGroupLayout Component."

- af:panelLabelAndMessage
- af:separator
- af:showDetailItem
- af:showPopupBehavior

  For information about how to use the af:showPopupBehavior component, see Section 29.5.3, "How to Configure a Hierarchy Viewer to Invoke a Popup Window."
- af:spacer
- dvt:panelCard

  For more information about how to use the dvt:panelCard component, see Section 29.6, "Using Panel Cards."

Note: Unsupported components are flagged at design time.

To add a node to a hierarchy viewer component:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **Insert inside dvt:hierarchyViewer** > **Node**.

   The following entry appears in the JSF page:

   ```
   <dvt:node>
   <f:facet name="zoom100"/>
   </dvt:node>
   ```

2. In the Structure window, right-click the **dvt:node** and choose **Go to Properties**.
3. Configure the appropriate properties in the Property Inspector.

   For example, set a value for the type property to associate a node component with an accessor:

   ```
   <dvt:node type="model.rootEmp model.HvtestView"/>
   ```

   For more information, see Section 29.3.3, "How to Specify a Node Definition for an Accessor."

29.3.2 How to Configure the Controls on a Node

The node component (dvt: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.

To configure the controls on a node:

1. In the Structure window, right-click the dvt:node component and choose **Go to Properties**.
2. Configure properties in the **Appearance** section of the Property Inspector for the dvt:node component, as described in Table 29–2.

Table 29–2 Node Configuration Properties

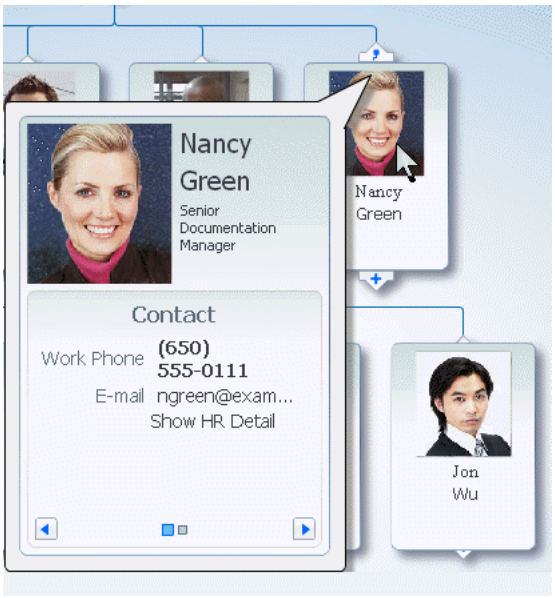
To do this:	Set the following value for this property:
Configure the Hide or Show controls to appear or not on a node.	Set `showExpandChildren` to `False` to hide the controls. By default the property is set to `True`.
Configure the Restore or Isolate controls to appear or not on the node.	Set the `showNavigateUp` and `showIsolate` properties to `False` to hide these controls on the node. By default the property is set to `true`. If the `showNavigateUp` property is set to `true`, you must also set a value for the hierarchy viewer component's `navigateUpListener` property, as described in Section 29.4.1, "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 Property Inspector for the node component, see Section 20.4, "Changing the Style Properties of a Component."

The hover detail window is automatically displayed when the user moves the mouse over the node, reflecting the `shape` attribute set for the node. A node with the `shape` attribute `roundedRect`, for example, will have a detail window with the same attribute, as shown in Figure 29–8.

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 29.7.5, "How to Disable the Hover Detail Window."

Figure 29–8 Hover Window in Hierarchy Viewer Node



29.3.3 How to Specify 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 Viewers" section in the *Oracle Fusion Middleware Fusion Developer's Guide for 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, "How to Associate 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 checks for a value of the `rendered` property that evaluates to `true`. The result of evaluating the `rendered` property does not affect the `type` property.

29.3.4 How to Associate 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.

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.

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 Property Inspector.
3. Specify the number of levels of ancestor nodes that you want to appear at runtime above the anchor node in the `displayLevelsAncestor` property.

   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 Navigating 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."

29.4.1 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.

To configure upward navigation for a hierarchy viewer component:

1. In the Structure window, select **dvt:hierarchyViewer** and choose **Go to Properties**.
2. In the Property Inspector, expand the **Behavior** section of the Property Inspector and write a value 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.
3. Click **OK**.

29.4.2 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

To configure same-level navigation in a hierarchy viewer component:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and select **Go to Properties**.
2. Expand the **Common** section of the Property Inspector 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. Click **OK**.

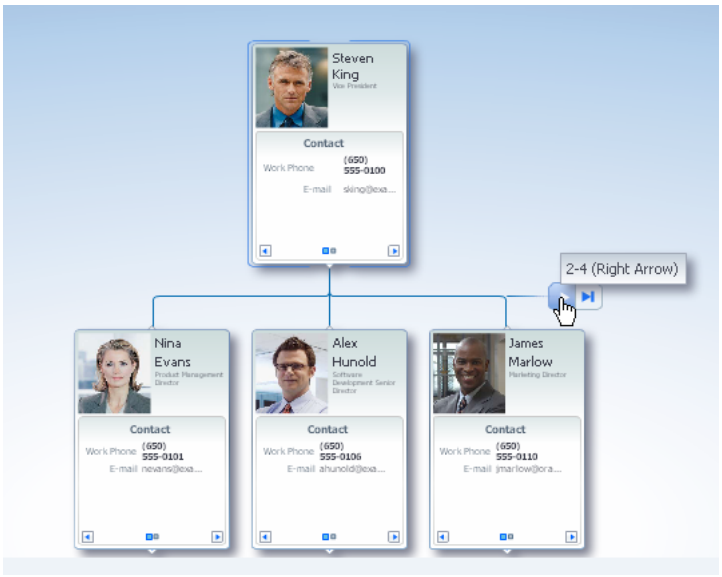
29.4.3 What Happens When You Configure Same-Level Navigation in a Hierarchy Viewer

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 29–9 shows a runtime example where `levelFetchSize="3"`. When a user moves the mouse over the control, the control that allows users to navigate to the last set of nodes appears.

Figure 29–9 Hierarchy Viewer Component with Same-Level Navigation



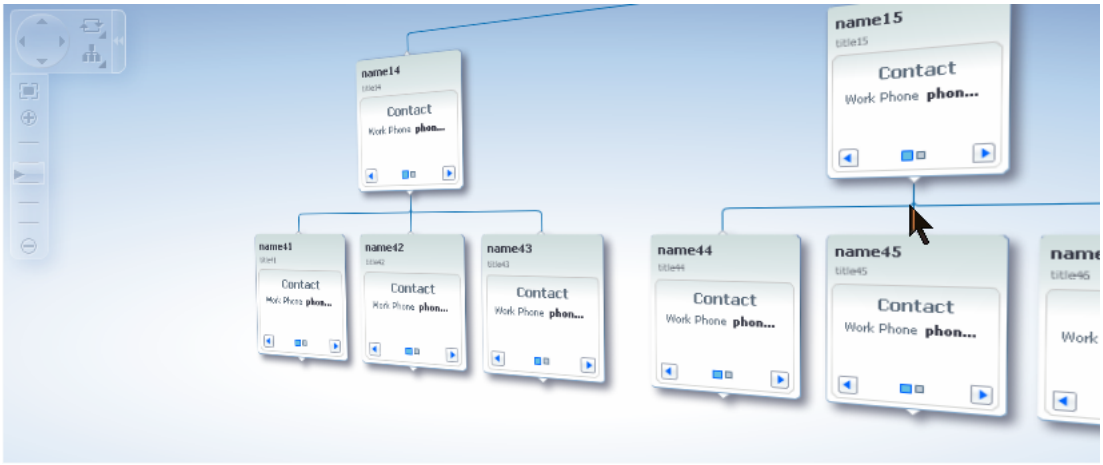
29.5 Adding Interactivity to a Hierarchy Viewer Component

You can configure a hierarchy viewer component to invoke popup windows and display menus with functionality and data from other pages in your Oracle Fusion web application.

29.5.1 How to Configure 3D Tilt Panning

By default, panning in a hierarchy viewer is accomplished by clicking and dragging the component to reposition the view, or by using the panning control in the Control Panel. In a hierarchy viewer with a large quantity of nodes, instead of browsing through a hierarchy viewer one page at a time, users can initiate a 3D tilt panning effect that animates the hierarchy viewer to visually fly 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 29–10 shows the tilt panning effect as it reaches the edge of the view.

Figure 29–10 Hierarchy Viewer 3D 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 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 cursor within the view to change the direction of the pan through the view. Exit tilt panning by selecting any node in the view.

You configure 3D tilt panning effect for the hierarchy viewer by setting the `panning` property to `tilt`.

29.5.2 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.

29.5.3 How to Configure 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:commandButton`. 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 29–11 shows a modal popup invoked from an **HR Detail** link in the node. Example 29–2 shows sample code for creating the popup. Example 29–3 shows sample code for the invoking the popup from a command component. For brevity, elements such as `<af:panelGroupLayout>`, `<af:spacer>`, and `<af:separator>` are not included in the sample code.

Figure 29–11 Modal Popup in Hierarchy Viewer Node

Example 29–2 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 29–3 Sample Code to Invoke Popup from Command Component

```
<f:facet name="zoom100">
...
<dvt:panelCard effect="slide_horz"
...
<af:showDetailItem text="Contact "
...
<af:commandLinktext="ShowHRDetail"inlineStyle="font-size:14px;color:#383A47">
  <af:showPopupBehavior popupId="::popupDialog" triggerType="action"
              align="endAfter" alignId="pg1" />
</af:commandLink>
</showDetailItem>
```

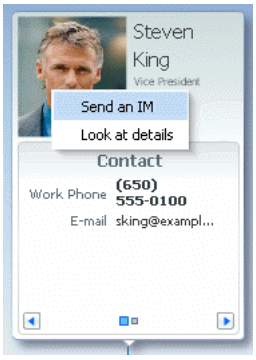
```
</dvt:panelCard>
</f:facet>
```

For more information about using the `af:showPopupBehavior` tag, see Section 13.4, "Invoking Popup Elements."

29.5.4 How to Configure a Hierarchy View Node to Invoke a Menu

You can configure a node component (`dvt:node`) within a hierarchy viewer to invoke a menu by using the `af:menu` component. You can configure one or more `af:commandMenuItem` elements for the `af:menu` component. Nodes within a hierarchy viewer component do not support the nesting of `af:menu` components. Figure 29–12 shows a context menu associated with a node.

Figure 29–12 Hierarchy Viewer Node Context Menu



Example 29–4 shows sample code for creating the context menu

Example 29–4 Context Menu Sample Code

```
<af:popup id="popupDialog" contentDelivery="lazyUncached" >
 <af:menu>
    <af:commandMenuItem text="Send an IM"/>
    <af:commandMenuItem text="Look at details"/>

  </af:menu>
</af:popup>
```

For more information about using the `af:menu` component, see Chapter 13, "Using Popup Dialogs, Menus, and Windows."

29.6 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.

29.6.1 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.

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. Select **Insert inside f:facet -zoom100** > **Panel Card**.

3. Use the Property Inspector to configure the properties of the panel card component.

   For example, set a value for the **Effect** property in the Advanced properties for the panel card component. Valid values are:

   - `Horizontal Slide (default)`
   - `Panel Flip`
   - `Node Flip`
   - `No Animation`

4. In the Structure window, right-click **dvt:panelCard** and choose **Insert inside dvt:panelCard** > **Show Detail Item**.

5. Repeat Step 4 as necessary.

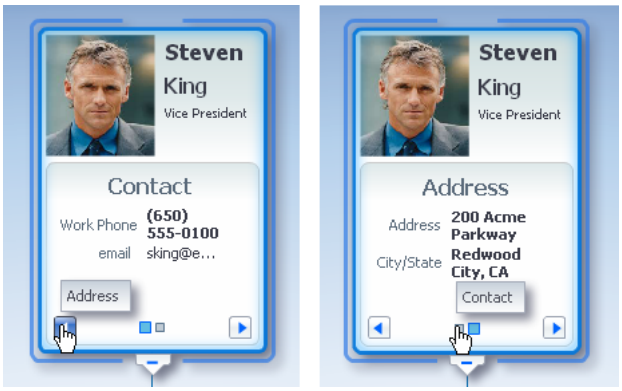
6. Use the Property Inspector to configure the properties of the `af:showDetailItem` elements that you insert.

29.6.2 What Happens at Runtime When a 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



29.7 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 20, "Customizing the Appearance Using Styles and Skins."

29.7.1 How to Adjust the Size of a Hierarchy Viewer

You can adjust the size of the hierarchy viewer component by setting values for a number of the hierarchy viewer component's attributes using the Property Inspector.

To adjust the size of a hierarchy viewer:

1. In the Structure window for the JSF page that contains the hierarchy viewer component, select the **dvt:hierarchyViewer** node.
2. In the Property Inspector, write 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 the `InlineStyle` property must use this syntax:

   ```
   width:100%;height:600px;
   ```
3. Save changes to the JSF page.

29.7.2 How to Include 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}"
inlineStyle="width:64px;height:90px;"/>
</af:panelGroupLayout>
```

For more information about the `af:panelGroupLayout` component, see Section 8.12.1, "How to Use the panelGroupLayout Component."

29.7.3 How to Configure the Display of the Control Panel

You can configure the hierarchy viewer component so that the Control Panel described in Section 29.1.2, "Hierarchy Viewer Elements and Terminology," acts 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

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. Expand the **Appearance** section and select 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`

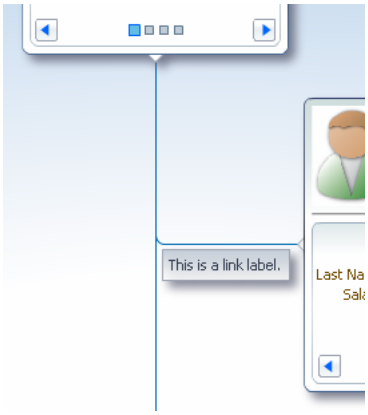
     This is the default value. The Control Panel appears in a collapsed or hidden state at runtime.
   - `initExpanded`

     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.

29.7.4 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 29–14 illustrates links and labels in a hierarchy viewer.

Figure 29–14 Hierarchy Viewer Links and Labels



You can customize the appearance of links and labels by setting properties of the `dvt:link` element in a hierarchy viewer. Figure 29–15 illustrates links with a `dashDot` value set for the `linkStype` attribute.

Figure 29–15 Links with dashDot Link Style

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 Property Inspector, 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 Property Inspector, 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}"
   ```

4. Optionally, also in the Property Inspector, 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"}
```

29.7.5 How to Disable the Hover Detail Window

By default, the hover window automatically displays when the zoom level is at 76-100%. 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 when the zoom level is at 76-100%.

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 Property Inspector, 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 displayed.
   - `none`

     Select this value if you do not want the hover window to display when the zoom level is at 76-100%.

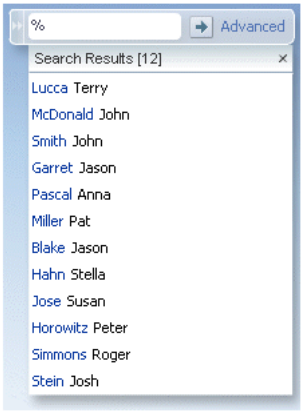
29.8 Adding Search to a Hierarchy Viewer

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 29–16 shows a sample search panel.

Figure 29–16 Hierarchy Viewer Search Panel

Figure 29–17 shows sample search results.

Figure 29–17 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.

To configure search in a hierarchy viewer:

1. In the Structure window, right-click the **dvt:hierarchyViewer** node and choose **insert inside dvt:hierarchyViewer** > **dvt:search**.
2. In the Property Inspector, 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, add a `f:facet` with a value of `name="end"` to specify a component that will launch an advanced search outside of the hierarchy viewer component. This facet should contain only a single component, for example `af:commandLink`, to launch a comprehensive search of a data set. For more information, see Section 12.4, "Using the query Component."
4. In the Structure window, right-click the **dvt:search** node and choose **insert inside dvt:search** > **dvt:searchResults**.
5. In the Property Inspector, 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 select **insert inside dvt:searchResults**, then select **ADF Faces**, then select **af:setPropertyListener**.
7. In the Property Inspector, 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, right-click the **dvt:searchResults** node and choose **insert inside dvt:searchResults** > **f:facet** with a value of `name="content"`.
9. In the Structure window, right-click the **f:facet** `content` node and do the following to specify the components to stamp out the search results:
   - Insert an ADF Faces `af:panelGroupLayout` element to wrap the output of the search results.
   - Insert the ADF Faces output components to display the search results. For example:

     ```
     <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 29–5 shows sample code for configuring search in a hierarchy viewer.

Example 29–5 Sample Hierarchy Viewer Search Code

```
<dvt:hierarchyViewer>
  <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 inlineStyle="width:110px;height:20px;">
        <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

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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Part V

Advanced Topics

Part V contains the following chapters:

30

Creating Custom ADF Faces Components

This chapter describes how to create custom ADF Faces rich client components.

This chapter includes the following sections:

30.1 Introduction to 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 standard JSF components is covered in many books, articles, web sites, and the JavaServer Faces specification, therefore, it is not covered in this guide. This chapter is intended to describe 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 3, "Using ADF Faces 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 Rich Client."

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 20, "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 `FusionFX`, `blafplus-rich`, and `blafplus-medium` skins inherit 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.

30.1.1 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 1.2.1.2, "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 30–1 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 30–1 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 30–1 (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`

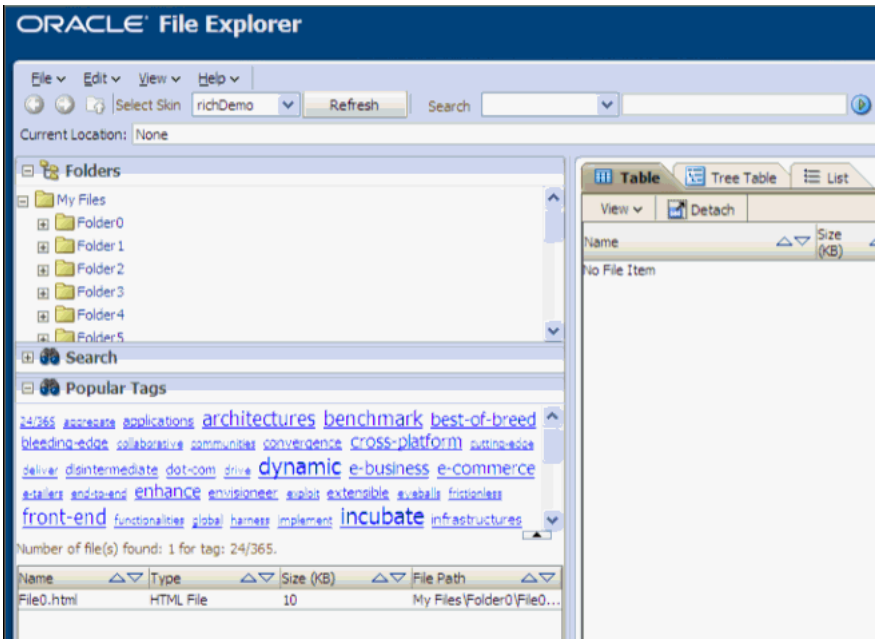
	Resource Bundle:
	`com.<component_package>.faces.resource.<Bundle_name>Bundle.java`
	Extends:
	`java.util.ListResouceBundle.java`

30.1.2 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 30–1 shows how the `tagPane` component would be displayed if it was added below the Search pane in the File Explorer application.

Figure 30–1 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 30–1 shows how the `tagPane` component might be used in a JSF page.

Example 30–1 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 30–2 shows a sample set of style selectors in the CSS file for the `tagPane` component.

Example 30–2 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 30–3 shows HTML server-side code used for the `tagPane` component.

Example 30–3 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 3, "Using ADF Faces 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 30–2 lists the client-side and server-side artifacts for the `tagPane` component. Referencing the naming conventions in Table 30–1, the *`component_package`* is `com.adfdemo.acme` and the *`prefix`* is `Acme`.

Table 30–2 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 30–2 (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`

30.2 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` folder 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 30.3, "Client-Side Development," and Section 30.4, "Server-Side Development."

30.2.1 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 2, "Getting Started with ADF Faces." For more information about deployment, see the "Deploying Fusion Web Applications" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

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 2.2, "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 Application Navigator, right-click the project and choose **New**.
   b. In the New Gallery, select **Deployment Profile** 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 1.2**, and **JSP 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 30–3.

   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 30–3 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`

30.2.2 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.

To create a faces-config.xml file for the custom component:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General**, select **XML**, and then Select **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.

30.2.3 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 Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **XML**.
3. Select **XML Document from XML Schema** and click **OK**.
4. 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**.
5. 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.

30.2.4 How to Add a Cascading Style Sheet

Add a cascading style sheet to define component's style.

To create a cascading style sheet for the custom component:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General,** 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`.

30.2.5 How to Add a Resource Kit Loader

Create an empty file and add the fully qualified classpath to the custom resource loader.

To create a resource loader for the custom component:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and then **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.

30.2.6 How to Add a JavaServer Pages Tag Library Descriptor File

You need a JSP TLD file to work with JSF pages.

To create a JavaServer Pages TLD file for the custom component:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **Web Tier** and select **JSP**.
3. Select **JSP Tag Library** and click **OK**.
4. In the Create JavaServer Page Tag Library dialog, select **Deployable** and click **Next**.
5. Enter the following:
   - Tag Library Descriptor Version: Select **2.1**.

- **Short Name**: A name. For example, for the **tagPane** component, you would enter `acme`.
- **Tag Library URI**: A URI for the tag library. For example, for the **tagPane** component, you would enter `http://oracle.adfdemo.acme`.

6. Click **Next** and optionally enter additional tag library information, then click **Finish**.

30.2.7 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.

To create an adf-js-features.xml file for the custom component:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **XML**.
3. In the right pane, select **XML Document from XML Schema** and click **OK**.
4. 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**.
5. 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.

30.2.8 How to Add a Facelets Tag Library Configuration File

If a consuming application uses Facelets, then you must define the handler for the component.

To create a Facelets tag library file:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **XML**.
3. In the right pane, select **XML Document** and click **OK**.
4. 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.
5. Copy and paste the code shown in Example 30–4:

Example 30–4 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>
```

6. Replace the namespace and tag-name code shown in bold with code appropriate for your application.

30.3 Client-Side Development

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 30.4, "Server-Side Development."

> **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 3.1, "Introduction to 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 30.2, "Setting Up the Workspace and Starter Files."

30.3.1 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.

To create the component JavaScript file:

1. In the Application Navigator, right-click the project and click **New**.
2. In the New Gallery, expand **Web Tier** and select **HTML**.
3. Select **JavaScript File** and click **OK**.
4. 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`.
5. Open the JavaScript File in the editor and add the component code to define the component type. Example 30–5 shows the code that might be used for the `tagPane` component.

Example 30–5 tagPane Component JavaScript

```
AdfUIComponents.createComponentClass(
```

```
 "AcmeTagPane",
 {
      componentType:"oracle.adfdemo.acme.TagPane",superclass:AdfUIObject
 }
);
```

30.3.2 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.

To create the JavaScript for the event:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **Web Tier** and select **HTML**.
3. Select **JavaScript File** and click **OK**.
4. 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`.
5. Open the JavaScript File in the editor and add the event code. Example 30–6 shows the event code that might be added for the `tagPane` component.

Example 30–6 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;
}
```

30.3.3 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.

To create the peer JavaScript file:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **Web Tier** and select **HTML**.
3. Select **JavaScript File** and click **OK**.
4. 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`.
5. Open the JavaScript file in the editor and 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 30–7 shows the code that might be added for the `tagPane` component.

Example 30–7 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");
```

30.3.4 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. Follow the procedures documented in Section A.9.1, "How to Create a JavaScript Feature," omitting the steps for creating the XML files, as you have already done so. Example 30–8 shows the `adf-js-features.xml` file used for the `tagPane` component.

Example 30–8 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>
```

30.4 Server-Side Development

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 30.2, "Setting Up the Workspace and Starter Files."

30.4.1 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.

To create the EventListener class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **Java**.
3. Select **Java Interface** and click **OK**.
4. 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`.
5. Open the Java file in the editor and 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 30–9 shows the code for the `tagPane` event listener.

Example 30–9 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;
}
```

30.4.2 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 30.3.2, "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.

To create the server-side event class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **Java**.
3. Select **Java Class** and click **OK**.
4. 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 30–10 shows the code for the event class.

Example 30–10 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;
}
}
```

30.4.3 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 rich client 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 30–4.

Table 30–4 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.
Rich Client 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.
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.

30.4.4 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.

To create the component class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **Java**.
3. Select **Java Class**. Click **OK**.
4. 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**.
5. 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);
   ```

6. 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 30–11 shows the `PropertyKey` methods for the `tagPane` component.

Example 30–11 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);
```

7. 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 30–12 shows the code after replacing the private attribute.

Example 30–12 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);
}
```

8. 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 30–13.

Example 30–13

```
/**
 * <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 ADF Faces JavaDoc 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 4, "Using the JSF Lifecycle with ADF Faces."

Example 30–14 shows the overwritten `broadcast` method for the `tagPane` component.

Example 30–14 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);
}
```

30.4.5 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`.

To register a custom component:

1. In the Application Navigator, double-click the `faces-config.xml` file.
2. Click the **Overview** tab and click the **Components** navigation tab.
3. Click the **Add** icon and enter the type and class for the component.
4. Optionally, add any attributes, properties, or facets.

Example 30–15 shows the `tagPane` component defined within a `faces-config.xml` file.

Example 30–15 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>
```

30.4.6 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 about resource bundle classes, see Section 20.3, "Defining Skin Style Properties."

> **Tip:** You can also use a properties file for your resources.

To create the resource bundle class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **Java**.
3. Select **Java Class** and click **OK**.
4. 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**.

5. Add any keys and define the text as needed. For more information about creating resource bundles for skins, see Section 20.3.1, "How to Apply Skins to Text."

   Example 30–16 shows the resource bundle code for the `tagPane` component.

Example 30–16 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}"}
   };
 }
}
```

6. 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 Property Inspector, 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 Property Inspector, 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 30–17 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 30–17 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>
```

30.4.7 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.

To create the renderer class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** then select **Java**.
3. Select **Java Class** and click **OK**.
4. 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**.
5. Add any needed functionality. For example, 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`.

30.4.8 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.

To register the render kit and renderer:

1. In the Application Navigator, double-click the `faces-config.xml` file to open it in the editor.
2. 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 30.4.7, "How to Create a Class for a Renderer." For example, for the `tagPane` component, you would enter `oracle.adfdemo.acme.faces.render.TagPaneRenderer`.

Example 30–18 shows the registration of the `tagPane` component render kit and renderer.

Example 30–18 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>
```

30.4.9 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 30.2.8, "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.

To add a JSP tag:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** then select **Java**.
3. Select **Java Class** and click **OK**.
4. 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**.

5. In the source editor, add all the attributes to the file.

   Example 30–19 shows the code for the attributes for the `TagPaneTag` class.

Example 30–19 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;
```

6. To declaratively generate the accessor methods for the attributes, right-click the file in the source editor and choose **Generate Accessors**.

7. In the Generate Accessors dialog, click **Select All**, set the **Scope** to `public` and click **OK**.

8. 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 30–20 shows the code for the `TagPaneTag` class, where both the component type and render type are `oracle.adfdemo.acme.TagPane`.

Example 30–20 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";
```

9. 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 30–21 shows the overridden method for the `tagPaneTag` class.

Example 30–21 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);

}
```

30.4.10 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:

Associate the tag library with a URI, assign a version, and give it a name. You should have already performed this step when you created the tag library stub file in Section 30.2.6, "How to Add a JavaServer Pages Tag Library Descriptor File."

To configure the TLD:

1. Open the skeleton TLD file.
2. In the Component Palette, 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 Property Inspector, 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.

Example 30–22 shows the TLD for the `tagPane` component.

Example 30–22 tagPane acme.tld Tag Library Descriptor Code

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<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>
```

30.4.11 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 into the class path of their web project and add a few entries into their web deployment descriptor file (`web.xml`). The rich client 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.

To create a resource loader class:

1. In the Application Navigator, right-click the project and select **New**.
2. In the New Gallery, expand **General** and select **Java**.
3. Select **Java Class** and click **OK**.
4. 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**.
5. 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 30–23 shows the expression for the `tagPane` component that maps the `/acme/images/` directory located relative to the `/META-INF` folder of the custom component JAR. As a result of the registration, the custom component images should be located under `/META-INF/acme/images`.

Example 30–23 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"));
```

6. 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
   ```

30.4.12 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.

ADF Faces provides three skins for use in your applications:

- `blafplus-rich`: Defines the default styles for ADF Faces components. This skin extends the `blafplus-medium` skin.
- `blafplus-medium`: Provides a modest amount of styling. This style extends the `simple` skin.
- `simple`: Contains only minimal formatting.

Skins provide more options than setting standard CSS styles and layouts. The skin's CSS file is processed by the skin framework to extract skin properties and icons and register them with the `Skin` object. Style sheet rules include a style selector, which identifies an element, and a set of style properties, which describes the appearance of the components.

All ADF Faces components use skins. The default skin is the simple 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 rich and medium skins inherit 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.

To create styles for your component:

1. Open the CSS file you created in Section 30.2.4, "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 30–24 shows the CSS file for the `tagPane` component.

Example 30–24 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 Section 20.3, "Defining Skin Style Properties."

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 Property Inspector, 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 20, "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 30–25 show the code for the `tagPane` component.

Example 30–25 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`.

30.5 Deploying a Component Library

After creating the custom component library, you must create a deployable artifact that can be used by a web application. Before you can build a Java archive (JAR) file, update the project's deployment profile by adding the many resources you created.

To create the JAR file for deployment:

1. In the Application Navigator, double-click the project to open the Project Properties dialog.
2. 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 select **Deploy** >***Project_name*** from the context menu. By default, the JAR file will be deployed to a deployment directory within the project directory.

30.6 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 of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 Component Palette.

30.6.1 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.

To configure the web.xml file:

1. In the Application Navigator, double-click the **web.xml** file to open it.
2. In the overview editor, select the **Servlets** navigation tab, and click the **Add** icon to add a new servlet.
3. In the Servlets table, do the following:
   - **Name**: Enter `resources`.
   - **Servlet Class**: Enter `org.apache.myfaces.trinidad.webapp.ResourceServlet`.
4. Below the table, click the **Servlet Mappings** tab, then click the **Add** icon.
5. 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`.

30.6.2 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 rich client 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."

To turn on logging and assertion:

1. In the Application Navigator, 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. 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.

30.6.3 How to Add a Custom Component to JSF Pages

To add the custom component to a JSF page:

1. Open the jspx page in the source editor.
2. 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 Component Palette to add the component to the page. Use the Property Inspector 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>
   ```

30.6.4 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 30–26 shows the backing bean code that is used to bind the `tagPane` component to the File Explorer application.

Example 30–26 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);

 }
```

31

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 the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

This chapter includes the following sections:

- Section 31.1, "Introduction to User Customization"
- Section 31.2, "Implementing Session Change Persistence"

31.1 Introduction to 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 31–1 shows the changes by component that provide default personalization capabilities:

Table 31–1 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.
activeCommandToolbar Button commandButton commandImageLink commandLink commandMenuItem commandNavigationItem commandToolbarButton	windowHeight	When users change the contentHeight attribute value of a panelWindow or dialog component, any associated windowHeight value on a command component is also changed and will remain.

Table 31–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
activeCommandToolbarButton commandButton commandImageLink commandLink commandMenuItem commandNavigationItem commandToolbarButton	windowWidth	When users change the contentWidth attribute value of a panelWindow or dialog component, any associated windowWidth value on a command component is also changed and will remain.
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 10.2.4, "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 31–1 (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`.
`table`	`first`	This attribute represents the index of the first row in the current range of rows, and is used to control which range of rows to display to the user. The value of this attribute is persisted only in response to a `RangeChangeEvent` and only when in screen reader mode.

31.2 Implementing Session Change Persistence

In order for the application to persist user changes to the session, you must configure your project to enable customizations.

31.2.1 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 Application Navigator, 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**.

31.2.2 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 31–1 shows the context parameter in the `web.xml` file.

Example 31–1 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>
```

31.2.3 What Happens at Runtime

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. Every time the page is requested, in the subsequent view or restore view phase, 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.

31.2.4 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.jspx` and `pageTwo.jspx` 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.jspx` 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.jspx` page. If the user navigates to the `pageTwo.jspx` page, no attribute change is applied.

32

Adding Drag and Drop Functionality

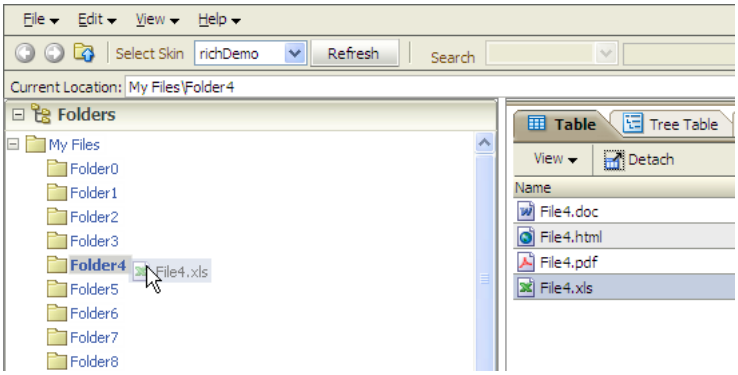
This chapter describes how to add drag and drop functionality to your pages, which allows users to drag the values of attributes or objects from one component to another, or allows users to drag and drop components.

This chapter includes the following sections:

32.1 Introduction to Drag and Drop Functionality

The ADF Faces framework provides the ability to drag and drop items from one place to another on a page. 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 32–1.

Figure 32–1 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 (Folder3). 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 of the `text` attribute of the `outputText` component becoming the value of the `text` 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 32–1.
- 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.

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). 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 (copy and paste as a link, for example, copying text and pasting the text as an actual URL).

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 32–1 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 32–1 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`

Table 32–1 (Cont.) Drag and Drop Scenarios

Scenario	Source	Target
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`
Dragging an object from a DVT Gantt chart and dropping it on another component	Gantt chart	Any component
	Tag: `dragSource`	**Tag**: `dropTarget`

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. Both the target and the source must contain the `dataFlavor` tag, and the values must be the same in order for the drop to be successful.

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.

32.2 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.

The following procedure assumes you have your target and source components already on the JSF page.

To add drag and drop functionality for attributes:

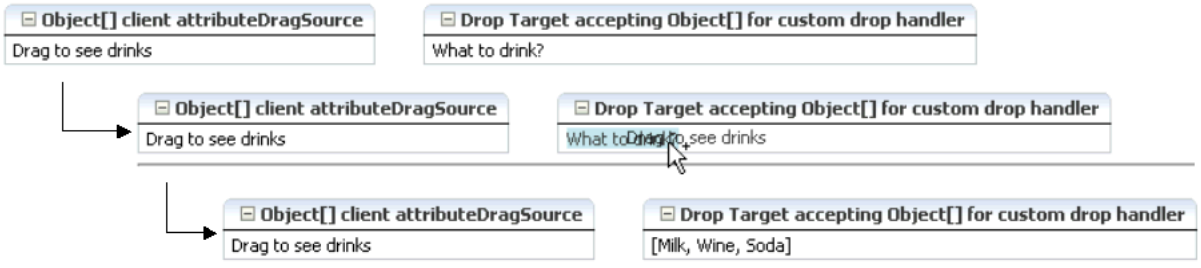
1. In the Component Palette, from the Operations panel, drag and drop an **Attribute Drop Target** as a child to the target component.
2. In the Insert Attribute Drop Target dialog, use the **Attribute** dropdown to select the attribute that will be populated by the drag and drop action. This dropdown list shows all valid attributes on the target component.
3. From the Component Palette, 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.

32.3 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 and you want the user to be able to drag the `outputText` component to a `panelBox` component and have that component display an `array`, as shown in Figure 32–6.

Figure 32–2 Dragging and Dropping an Array Object



The `outputText` component contains an `attributeDragSource` tag. However, because you want to drag an `array` (and not just the `String` value of the attribute), you must use the `dropTarget` tag instead of the `attributeDropTarget` tag. 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 32.9.1, "How to Add Drag and Drop Functionality for a DVT Component".

32.3.1 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.

This procedure assumes the source and target components already exist on the page.

To add drag and drop functionality:

1. In the JSF page that contains the target, add a `dropTarget` tag as a child to the target component by dragging and dropping a **Drop Target** tag (located in the Operations panel) from the Component Palette.
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).

   Tip: You can also intercept the drop on the client by populating the `clientDropListener` attribute. For more information, see Section 32.3.3, "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 Property inspector, 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 32–1 shows the code for a `dropTarget` component inserted into an `panelBox` component that takes an array object as a drop source. Note that because an action was not defined, the only allowed action will be `COPY`.

Example 32–1 JSP 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 returns 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 32–2 shows a private 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 32–2 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;
    }
  }
```

6. Add a clientAttribute tag as a child to the source component by dragging a **Client Attribute** (located in the Operations panel), from the Component Palette. This tag is used to define the payload of the source for the event. Define the following for the clientAttribute tag in the Property Inspector:
   - **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. Drag and drop an **Attribute Drag Source** (located in the Operations panel), from the palette as another child to the source component. 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 32–3 shows the code for an outputText component that is the source of the drag and drop operation.

Example 32–3 JSP 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>
```

32.3.2 What Happens at Runtime

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 5.4.3, "What You May Need to Know About Marshalling and Unmarshalling Data.".

32.3.3 What You May Need to Know About Using the ClientDropListener

The `dropTarget` tag contains the `clientDropListner` attribute where you can reference JavaScript that will handle the drop event on the client. The client handler should not take any parameters and returns 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 32–4 shows a client handler that uses the logger to print a message.

Example 32–4 clientDropListener Handler

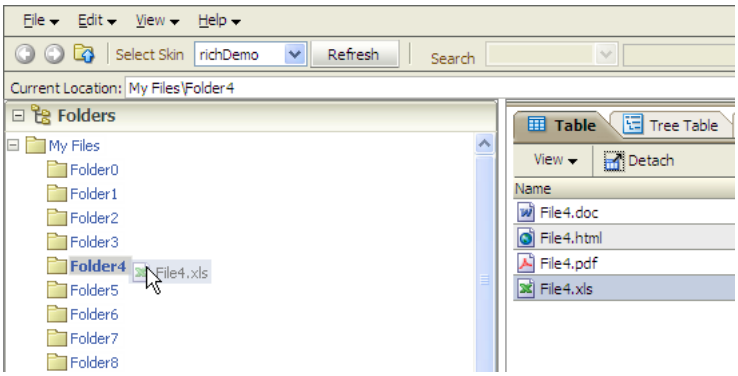
```
<script>
/**
 * Shows a message.
 */
function showMessage()
{
```

```
  AdfLogger.LOGGER.logMessage(AdfLogger.ALL, "clientDropListener handler,
    copying...");
  return AdfDnDContext.ACTION_COPY;
}
</script>
```

32.4 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 32–3 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 32–3 Drag and Drop Functionality in the File Explorer Application



As with dragging and dropping single objects, you can have a drop on a collection cause 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 32.4.2, "What You May Need to Know About the dragDropEndListener".

32.4.1 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.

This procedure assumes you already have the source and target components on the page.

To add drag and drop functionality:

1. Add a `collectionDropTarget` tag as a child to the target collection component by dragging a **Collection Drop Target** from the Component Palette.

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 Property Inspector, 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 32–5 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 32–5 Event Handler Code for a dropListener for a Collection

```
public DnDAction handleDrop(DropEvent dropEvent)
{
  Transferable transferable = dropEvent.getTransferable();

  // The data in the transferable is the row key for the dragged component.
  DataFlavor<RowKeySet> rowKeySetFlavor =
                      DataFlavor.getDataFlavor(RowKeySet.class, "DnDDemoModel");
  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.
      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().add(dnDDemoData);
    }
    return dropEvent.getProposedAction();
  }
  else
  {
    return DnDAction.NONE;
  }
}
```

5. In the Component Palette, from the Operations panel, drag and drop a **Drag Source** as a child to the source component.
6. With the `dragSource` tag selected, in the Property Inspector set the allowed Actions and any needed discriminant, as configured for the target.

32.4.2 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 32–6 shows a handler for a `dragDropEndListener`. attribute

Example 32–6 Handler for dragDropEndListener

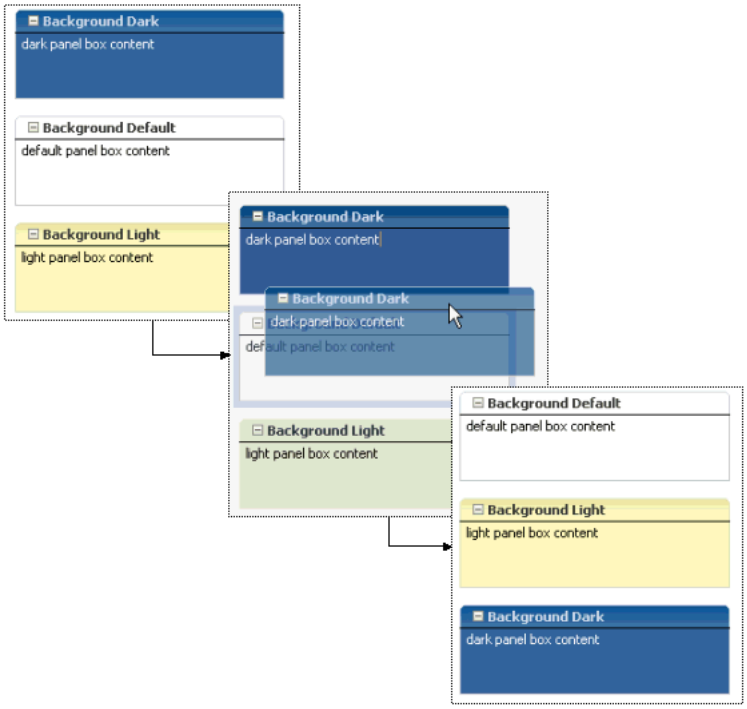
```
public void endListener(DropEvent dropEvent)
{
  Transferable transferable = dropEvent.getTransferable();

  // 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 dta from the source model directly.
    Object removed = getSource2Values().remove(currKey.intValue());
  }
  // Need to add the drag source table so it gets redrawn.

AdfFacesContext.getCurrentInstance().addPartialTarget(dropEvent.getDragComponent()
);
```

32.5 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 32–4

shows the darker `panelBox` component being moved from being the first child component of the `panelGrid` component to the last.

Figure 32–4 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 32.6, "Adding Drag and Drop Functionality Into and Out of a panelDashboard Component."

32.5.1 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.

To add drag and drop functionality:

1. From the Operations panel of the Component Palette, 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 4).
3. With the `dropTarget` tag still selected, in the Property Inspector, select a valid action set for the `action` 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.

   This method must take a `DropEvent` event as a parameter and return a `DnDAction`, 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 2.

   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 32–7 shows the `handleComponentMove` event handler on the `DemoDropHandler.java` managed bean used by the `componentDragSource` JSF page in the demo application.

Example 32–7 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;
    ComponentChange change;

    // 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))
      {
        reorderedIdList.add(currId);
        if (currId.equals(dropLayoutId))
        {
          reorderedIdList.add(movedLayoutId);
        }
      }
    }

    change = new ReorderChildrenComponentChange(reorderedIdList);
    rootParent = dropParent;
    // 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. Add a `componentDragSource` tag to the source component by dragging and dropping a **Component Drag Source** from the Component Palette as a child of the source component.

32.6 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.

32.6.1 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 cannot 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:

1. Create a `panelDashboard` component. For more information, see Section 8.7, "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 8.8.3, "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 Property Inspector, 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 Component Palette, from the Operations panel, 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 Property Inspector, 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 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 method must take a `DropEvent` event as a parameter and return a `DnDAction` of `NONE`, because the `panelDashboard` handles the positioning of its child components.

   This handler method should use the `dropEvent.getTransferable().getData(DataFlavor.UICOMPONENT_FLAVOR)` to get the `transferable` object and its data. Once the method completes the drop, you can use the `org.apache.myfaces.trinidad.change.ReorderChildrenComponentChange` method to preserve the new ordering of the children and the `dropEvent.getDropSiteIndex()` method to get the location at which the user wants the dragged component. You can also use the `dashboardComponent.prepareOptimizedEncodingOfInsertedChild()` method to animate the drop of the component.

   Example 32–8 shows the `move` event handler and helper methods on the `DemoDashboardBean.java` managed bean used by the `dashboard` JSF page in the ADF Faces demo application.

Example 32–8 Handler for DropListener on a panelDashboard Component

```
public DnDAction move(DropEvent e)
{
  UIComponent dropComponent = e.getDropComponent();
  UIComponent movedComponent = e.getTransferable().getData(DataFlavor.UICOMPONENT_FLAVOR);
  UIComponent movedParent = movedComponent.getParent();
   // Ensure that we are handling the re-order of a direct child of the panelDashboard:
  if (movedParent.equals(dropComponent) && dropComponent.equals(_dashboard))
  {
    // Move the already rendered child and redraw the side bar since the insert indexes have
    // changed:
    _moveDashboardChild(e.getDropSiteIndex(), movedComponent.getId());
  }
  else
  {
    // This isn't a re-order but rather the user dropped a minimized side bar item into the
    // dashboard, in which case that item should be restored at the specified drop location.
    String panelKey = _getAssociatedPanelKey(movedComponent);
    if (panelKey != null)
    {
      UIComponent panelBoxToShow = _dashboard.findComponent(panelKey);
        // Make this panelBox rendered:
        panelBoxToShow.setRendered(true);
```

```
      int insertIndex = e.getDropSiteIndex();

      // Move the already rendered child and redraw the side bar since the insert indexes have
      // changed and because the side bar minimized states are out of date:
      _moveDashboardChild(insertIndex, panelKey);

      // Let the dashboard know that only the one child should be encoded during the render phase:
      _dashboard.prepareOptimizedEncodingOfInsertedChild(
        FacesContext.getCurrentInstance(),
        insertIndex);
    }
  }

  return DnDAction.NONE; // the client is already updated, so no need to redraw it again
}

  private void _moveDashboardChild(int dropIndex, String movedId)
  {
    // Build the new list of IDs, placing the moved component at the drop index.
    List<String> reorderedIdList = new ArrayList<String>(_dashboard.getChildCount());
    int index = 0;
    boolean added = false;

    for (UIComponent currChild : _dashboard.getChildren())
    {
      if (currChild.isRendered())
      {
        if (index == dropIndex)
        {
          reorderedIdList.add(movedId);
          added = true;
        }

        String currId = currChild.getId();
        if (currId.equals(movedId) && index < dropIndex)
        {
          // component is moved later, need to shift the index by 1
          dropIndex++;
        }

        if (!currId.equals(movedId))
        {
          reorderedIdList.add(currId);
        }
        index++;
      }
    }

    if (!added)
    {
      // Added to the very end:
      reorderedIdList.add(movedId);
    }

    // Apply the change to the component tree immediately:
    ComponentChange change = new ReorderChildrenComponentChange(reorderedIdList);
    change.changeComponent(_dashboard);

    // Add the side bar as a partial target since we need to redraw the state of the side bar items
```

```
    // since their insert indexes are changed and possibly because the side bar minimized states
    // are out of date:
    RequestContext rc = RequestContext.getCurrentInstance();
    rc.addPartialTarget(_sideBarContainer);
  }
```

7. In the Component Palette, from the Operations panel, drag a **Component Drag Source** and drop it as a child to the `panelBox` component that will be the source component.
8. In the Property Inspector, set **Discriminant** to be the same value as entered for the **Discriminant** on the `panelDashboard` in Step 5.

32.6.2 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.

How to add drag and drop functionality out of a panelDashboard component:

1. In the Component Palette, from the Operations panel, 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 Property Inspector, select **MOVE** as the value `action` attribute.
4. In the Structure window, select the `dataFlavor` tag and in the Property Inspector, 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 32–9 shows the `handleSideBarDrop` event handler and helper methods on the `DemoDashboardBean.java` managed bean used by the `dashboard` JSF page in the demo application.

Example 32–9 Event Handler Code for a dropListener That Handles a panelBox Move Out of a panelDashboard Component

```
public DnDAction handleSideBarDrop(DropEvent e)
{
  UIComponent movedComponent = e.getTransferable().getData(DataFlavor.UICOMPONENT_FLAVOR);
  UIComponent movedParent = movedComponent.getParent();

  // Ensure that the drag source is one of the items from the dashboard:
  if (movedParent.equals(_dashboard))
```

```
  {
    _minimize(movedComponent);
  }

  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;
      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;

      _switcher.setFacetName("restored");
      rc.addPartialTarget(_switcher);
    }

    // Redraw the side bar so that we can update the colors of the opened items:
    rc.addPartialTarget(_sideBarContainer);
  }
```

6. In the Component Palette, 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 Property Inspector, set **Discriminant** to be the same value as entered for the **Discriminant** on the `dataFlavor` tag for the target component in Step 4.

32.7 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.

32.7.1 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.

To add drag and drop functionality to a calendar:

1. In the Component Palette, from the Operations panel, 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 Property Inspector, 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[]`.

32.7.2 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.

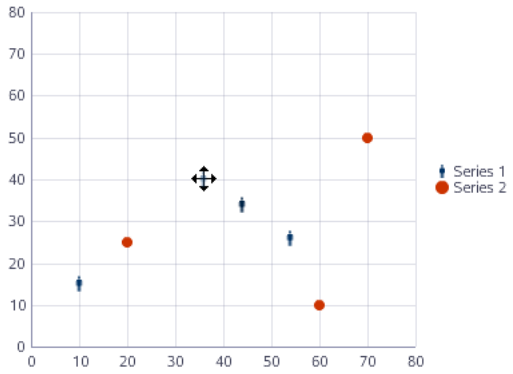
In the week view, users can move all-day and multi-day activities, however, they can be dropped only within other all-day slots. That is, the user cannot change an all-day activity to an activity with start and end times. In the month view, users can move all-day and multi-day activities to any other day.

32.8 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 GraphSelection object. You also must implement any required functionality in response to the drag and drop action.

For example, you might have a `scatterGraph` component and you want the user to be able to drag a human scatter marker to adjust the performance rating of an employee, as shown in Figure 32–6.

Figure 32–5 Dragging and Dropping an Object



The `scatterGraph` 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.

32.8.1 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 32.3.2, "What Happens at Runtime."

To add drag and drop functionality:

1. In the Component Palette, 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.GraphSelection`, 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 Property Inspector, 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 an `GraphSelection` object, GraphA and GraphB. You also have two sources, both of which are `GraphSelection` objects. By setting a discriminant value on GraphA with a value of `alpha`, only the `GraphSelection` source that provides the discriminant value of `alpha` will be accepted.
5. In the Structure window, select the `dropTarget` tag. In the Property inspector, select `MOVE` as the value for **Actions**.
6. In the Component Palette, from the Operations panel, drag and drop a **Drag Source** as a child to the graph component.
7. With the `dragSource` tag selected, in the Property Inspector set `MOVE` as the allowed Action and add any needed discriminant, as configured for the `dataFlavor` tag.
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.GraphSelection`.

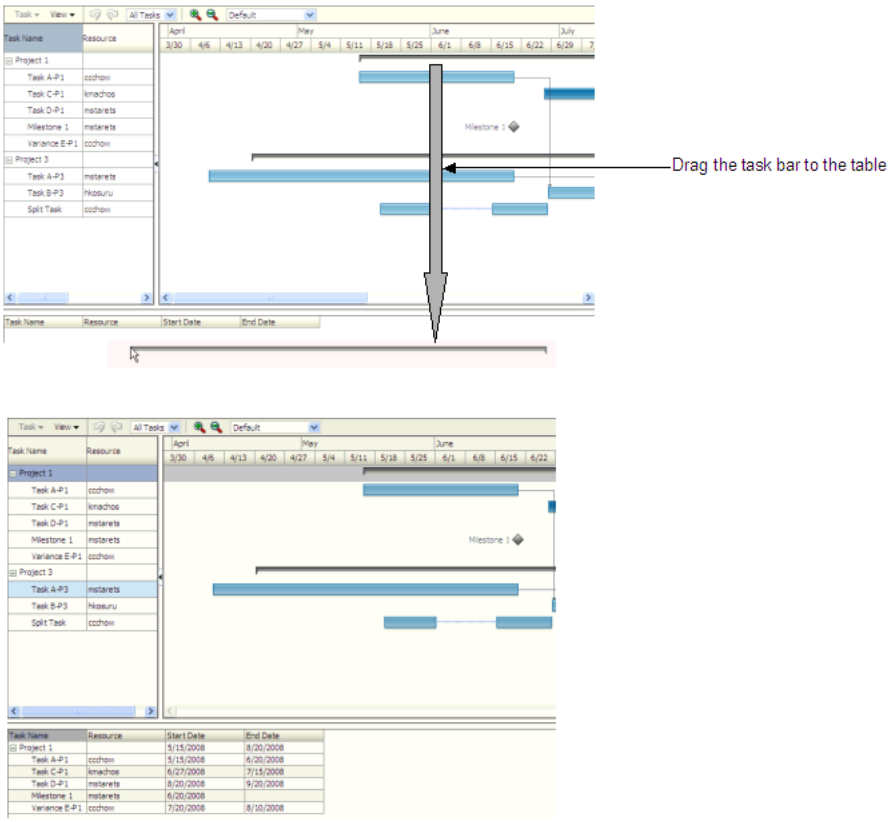
32.9 Adding Drag and Drop Functionality for DVT Gantt Charts

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 timeline to a `treeTable` component and have that component display information about the timeline, as shown in Figure 32–6.

Figure 32–6 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.

32.9.1 How to Add Drag and Drop Functionality for a DVT 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 32.3.2, "What Happens at Runtime." For information about using the `clientDropListener` attribute, see Section 32.3.3, "What You May Need to Know About Using the ClientDropListener."

To add drag and drop functionality:

1. In the Component Palette, 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 32.3.3, "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 Property Inspector, 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 Property inspector, 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 32–10 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 32–10 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 32–11 shows a handler method that copies a `TaskDragInfo` object from the event payload and assigns it to the component that initiated the event.

Example 32–11 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 Component Palette, 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 Property Inspector set the allowed Actions and any needed discriminant, as configured for the target.

33

Using Different Output Modes

This chapter describes how you can have your pages display in modes suitable for printing and emailing. Topics include how to use the `showPrintablePageBehavior` tag to print page contents and how to create emailable pages with the request parameter `org.apache.myfaces.trinidad.agent.email=true`.

This chapter includes the following sections:

33.1 Introduction to Using Different Output Modes

ADF Faces allows 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:goLink 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, allows users to view a simplified version of the 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 either Microsoft Outlook 2007 or Mozilla Thunderbird 2.0.

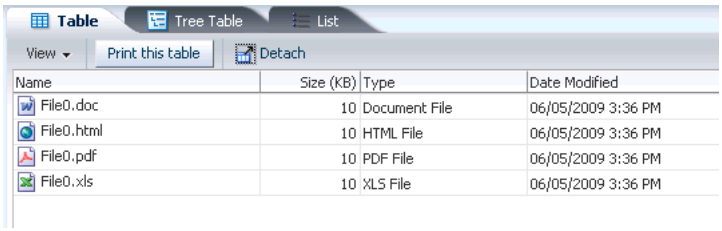
> **Tip:** The current output mode (`email` or `printable`) can be reached from `AdfFacesContext`. Because this context is EL reachable, you can use EL to bind to the output mode from the JSP page. For example, you might allow a graphic to be rendered only if the current mode is not `email` using the following expression:
>
> ```
> <af:activeImage source="/images/stockChart.gif"
>                 rendered="#{adfFacesContext.outputMode != "email"}"/>
> ```
>
> You can determine the current mode using `AdfFacesContext.getOutputMode()`.

33.2 Displaying a Page for Print

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` or a `panelAccordion` or the root of the tree (whichever comes first). The tree is rendered from there. Additionally, certain components that are not needed in print version (such as buttons, tabs, and scrollbars) are omitted.

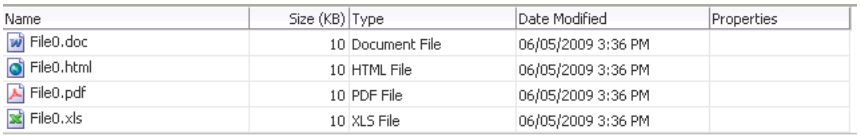
For example, in the File Explorer application, you could place a `commandButton` component inside the toolbar of the `panelCollection` component that contains the table, as shown in Figure 33–1.

Figure 33–1 Button to Print Part of a Page



When the user clicks the button, the page is displayed in a new browser window (or tab, depending on the browser) in a simplified form, as shown in Figure 33–2.

Figure 33–2 Printable Version of the Page



Only the contents of the table are displayed for printing. All extraneous components, such as the tabs, the toolbar, and the scroll bars, are not rendered.

When the button is clicked, the `action` event is canceled. Instead, a request is made to the server for the printable version of the page.

33.2.1 How to Use the showPrintablePageBehavior Tag

The `showPrintablePageBehavior` tag is used as a direct child of a command component.

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 18.2.1, "How to Use Command Buttons and Command Links."

   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 Component Palette, from the Operations panel, drag a **Show Printable Page Behavior** and drop it as a child to the command component.

33.3 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 we use 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 and Mozilla Thunderbird 2.0 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`
- `table`
- `column`
- `goLink` (renders as `text`)

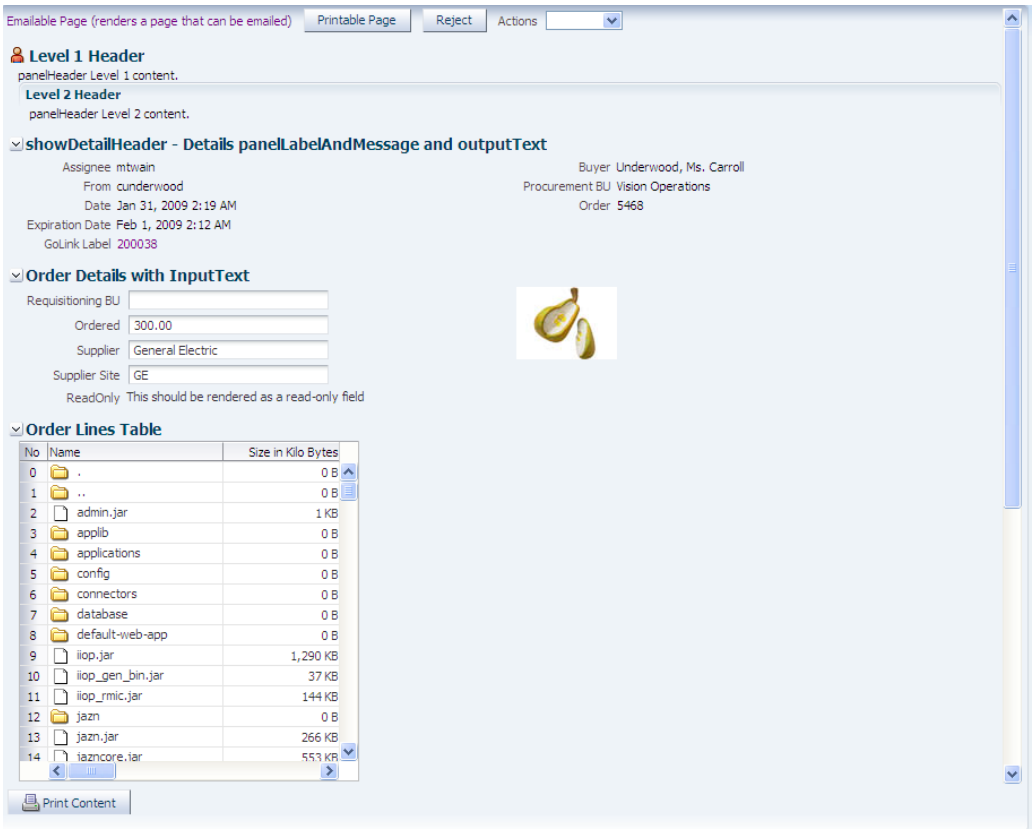
33.3.1 How to Create an Emailable Page

You notify the ADF Faces framework to convert your page to be rendered in an email client by appending the following the request parameter to the URL of the page to be emailed:

```
org.apache.myfaces.trinidad.agent.email=true
```

For example, say you have a page that displays a purchase order, as shown in Figure 33–3.

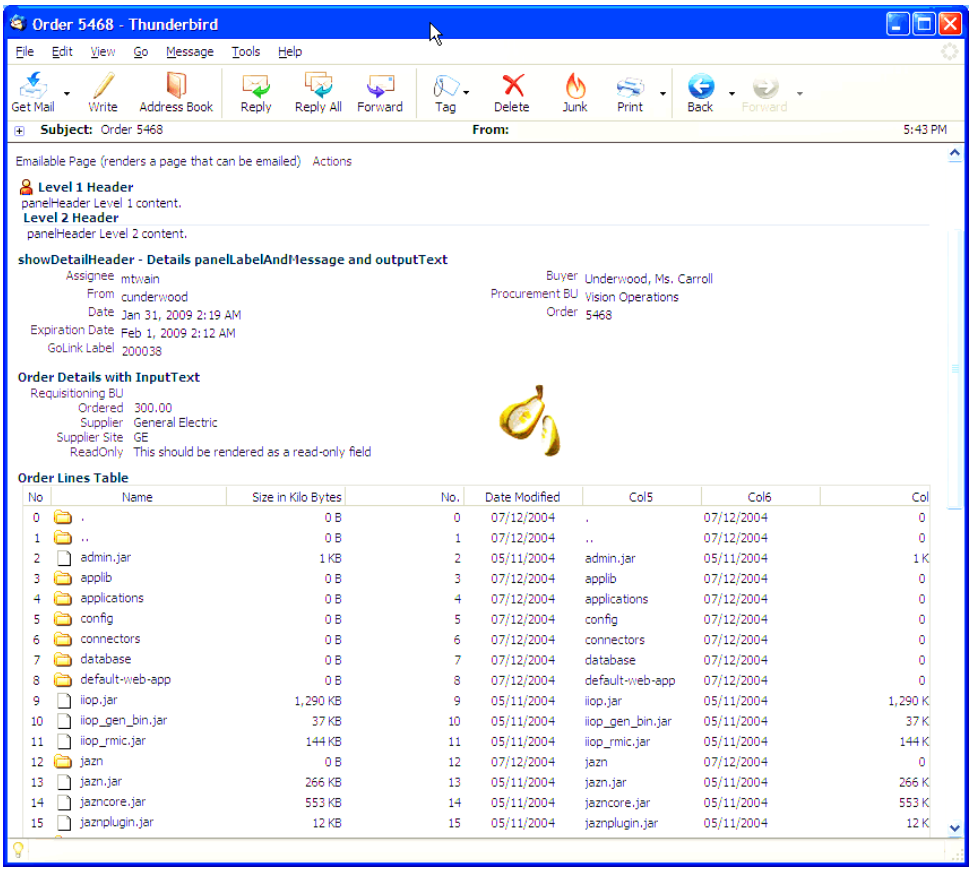
Figure 33–3 Purchase Order Web Page



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 33–4 shows the page as it appears in an email client.

Figure 33–4 Page in an Email Client



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.

33.3.2 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 `goButton` component.

To test an emailable page:

1. In the Component Palette, from the Common Components panel, drag and drop a **Go Button** anywhere onto the page.
2. In the Property Inspector, expand the Common section and 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
```

3. Right-click the page and choose **Run** to run the page in the default browser.

   The Configure 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 can be eight characters or more and must have a numeric character.

4. Once the page displays in the browser, click the `goButton` you added to the page. This will again display the page in the browser, but converted to a page that can be handled by an email client.

5. In your browser, view the source of the page. For example, in Mozilla Firefox, you would select **View > Page Source**. Select the entire source and copy it.

6. Create a new message in your email client. Paste the page source into the message and send it to yourself.

   Tip: Because you are pasting HTML code, you will probably need to use an insert command to insert the HTML into the email body. For example, in Thunderbird, you would choose **Insert > HTML**.

7. If needed, create a skin specifically for the email version of the page using an agent. The following example shows how you might specify the border on a table rendered in email.

```
af|table {
  border: 1px solid #636661;
}

@agent email {
  af|table
   {border:none}
}

af|table::column-resize-indicator {
  border-right: 2px dashed #979991;
}
```

   For more information about creating skins, see Chapter 20, "Customizing the Appearance Using Styles and Skins."

33.3.3 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
```

Part VI

Appendixes

Part VI contains the following appendixes:

- Appendix A, "ADF Faces Configuration"
- Appendix B, "Message Keys for Converter and Validator Messages"
- Appendix C, "Keyboard Shortcuts"
- Appendix D, "Creating Web Applications for Touch Devices Using ADF Faces"
- Appendix E, "Quick Start Layout Themes"
- Appendix F, "Troubleshooting ADF Faces"

A

ADF Faces Configuration

This appendix describes how to configure JSF and ADF Faces features in various XML configuration files, as well as how to retrieve ADF Faces configuration values using the `RequestContext` API and how to use JavaScript partitioning.

This chapter includes the following sections:

A.1 Introduction to Configuring ADF Faces

A JSF web application requires a specific set of configuration files, namely, `web.xml` and `faces-config.xml`. ADF 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 Rich Client."

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*.

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 JSP 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 Application Navigator, double-click **web.xml** to open the file.

   By default, JDeveloper opens the `web.xml` file in the overview editor, as indicated by the active **Overview** tab at the bottom of the editor window.

   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 `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, it is recommended that you 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.
- org.apache.myfaces.trinidad.USE_APPLICATION_VIEW_CACHE: Enables the *Application View Cache* (AVC), which can improve scalability by caching the state for the initial renders of the page's UI at an application scope. However, every page in the application must by analyzed for support in the AVC to avoid potential problems with debugging in an unexpected state and information leakage between users. Additionally, development is more difficult since page updates are not noticed until the server is restarted, and although initial render performance is enhanced, session size is not.

  CAUTION: The Application View Cache is not supported for this release. The feature does not work for any page where the rendering of the component tree causes the structure of the component tree to change temporarily. Since this is often the case, **USE_APPLICATION_VIEW_CACHE** should not be used.

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 JSPs 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`. Set to `INFO` if you have enabled automated profiler instrumentation code (see `oracle.adf.view.rich.profiler.ENABLED` in Section A.2.3.7, "Profiling").

  Performance Tip: JavaScript logging will affect performance. Set this value to `OFF` in a runtime environment.

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 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.2.3.6 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 "Enabling the Application for Real User Experience Insight and End User Monitoring" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

A.2.3.7 Profiling

You can specify the following JavaScript profiling context parameters:

- `oracle.adf.view.rich.profiler.ENABLED`: Specifies whether or not to use the automated profiler instrumentation code provided with the JavaScript Profiler. The default is `false`. Set to `true` to enable the JavaScript profile. When the profiler is enabled, an extra roundtrip is needed on each page to fetch the profiler data. By default, JDeveloper uses the `/WEB-INF/profiler.xml` configuration file. To override the location of the `profiler.xml` file, use the `ROOT_FILE` context parameter, as described next. You may also want to set `DEBUG_JAVASCRIPT` to `true`, to turn off JavaScript obfuscation. You also must set the `LOGGER_LEVEL` to at least `INFO`.
- `oracle.adf.view.rich.profiler.ROOT_FILE`: Specifies the initial `profiler.xml` file to load, if automated profiler instrumentation code is turned on. By default, JDeveloper uses the `/WEB-INF/profiler.xml` file if `ROOT_FILE` is not specified.

A.2.3.8 Facelets Support

Specify the following if you intend to use Facelets with ADF Faces:

- `org.apache.myfaces.trinidad.ALTERNATE_VIEW_HANDLER`: Install `FaceletsViewHandler` by setting the parameter value to `com.sun.facelets.FaceletViewHandler`

- `javax.faces.DEFAULT_SUFFIX`: Use `.xhtml` as the file extension for documents that use Facelets by setting the parameter value to `.xhtml`.

A.2.3.9 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.2.3.10 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.2.3.11 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 web sites. 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.2.3.16, "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:commandButton  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

Use the `oracle.adf.view.rich.tonalstyles.ENABLED` parameter to turn the use of tonal styles off or on. While the tonal style classes `.AFDarkTone`, `.AFMediumTone`, `.AFLightTone` and `.AFDefaultTone` are still available for the purpose of backward compatibility, themes are provided as a replacement style. 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. Set to `false` to disable tonal styles.

A.2.3.14 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 7.4, "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 5.1.1, "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 5.1.1, "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.2.3.15 JavaScript Partitioning

Use the `oracle.adf.view.rich.libraryPartitioning.ENABLED` parameter to turn JavaScript partitioning on and off. By default, the value is `true` (enabled). JavaScript partitioning allows a page to download only the JavaScript needed by client components for that page.

Valid values are:

- `true`: JavaScript partitioning is enabled (the default).
- `false`: JavaScript partitioning is disabled.

For more information about using and configuring JavaScript partitioning, see Section A.9, "Using JavaScript Library Partitioning."

A.2.3.16 Framebusting

Use the `oracle.adf.view.rich.security.FRAME_BUSTING` context parameter to use framebusting in your application. *Framebusting* is a way to prevent clickjacking, which occurs when a malicious web site 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 web site 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 `differentDomain`. This setting causes framebusting to occur only if the frame is in a page that originates from a different domain than your application. 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, 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 `differentDomain`, 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.
- `differentDomain`: 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: 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`

A.2.3.17 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.2.3.18 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 ADF 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–2 shows a sample ACF servlet definition.

Example A–2 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.2.3.19 Configuring Native Browser Context Menus for Command Links

Use the `oracle.adf.view.rich.ACTION_LINK_BROWSER_CONTEXT_SUPPRESSION` 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 Chapter 18.3, "Configuring a Browser's Context Menu for Command Links."

A.2.3.20 Session Timeout Warning

When a request is sent to the server, a session timeout value is written to the page and the session timeout warning interval is defined by the context parameter `oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_TIMEOUT`. The user is given the opportunity to extend the session in a warning dialog, and a notification is sent when the session has expired and the page is refreshed. Depending on the application security configuration, the user may be redirected to the log in page when the session expires.

Use the `oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_TIMEOUT` context parameter to set the number of seconds prior to the session time out when a warning dialog is displayed. If the value of `WARNING_BEFORE_TIMEOUT` is less than 120 seconds, if client state saving is used for the page, or if the session has been invalidated, the feature is disabled. The session time-out value it taken directly from the session.

Example A–3 shows configuration of the warning dialog to display at 120 seconds before the time-out of the session.

Example A–3 Configuration of Session Time-out Warning

```
<context-param>
    <param-name>oracle.adf.view.rich.sessionHandling.WARNING_BEFORE_
        TIMEOUT</param-name>
```

```
    <param-value>120</param-value>
</context-param>
```

The default value of this parameter is 120 seconds. To prevent notification of the user too frequently when the session time-out is set too short, the actual value of `WARNING_BEFORE_TIMEOUT` is determined dynamically, where the session time-out must be more than 2 minutes or the feature is disabled.

A.2.3.21 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.2.3.22 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

By default, this parameter is set to `on`. You can set it to `off`, so that the splash screen will not display.

A.2.3.23 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 absent. Valid values are `HTML5` and `FLASH`.

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 5 tutorial on Sun's web site (`http://java.sun.com`).

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 2.2, "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–4.

Example A–4 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 Task Flows" chapter of the *Oracle Fusion Middleware Fusion Developer's Guide for 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 Application Navigator, double-click **faces-config.xml** to open the file.

   By default, JDeveloper opens the `faces-config.xml` file in the overview editor, as indicated by the active **Overview** tab at the bottom of the editor window.

   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 ADF features that your application will use.

To create and edit adf-config.xml:

1. If not already created, create a `META-INF` directory for your project.
2. Right-click the `META-INF` directory, 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. 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–5.

Example A–5 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.2.3.18, "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–6 shows the syntax for defining caching rules in `adf-config.xml`.

Example A–6 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.

Table A–1 (Cont.) AFC Caching Rule Elements and Attributes

Rule Element Children	Attribute Description and Value

`<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–7 shows the syntax for application-wide disabling of Flash in `adf-config.xml`.

Example A–7 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.

Note: Previously Data Visualization `dvt:graph` and `dvt:gauge` components used an `imageFormat=AUTO"` value. The `AUTO` value has been deprecated and you should set use `imageFormat="FLASH"` and set `flash-player-usage` context parameter to `downgrade` to achieve the same effect application-wide.

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–8:

Example A–8 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, as shown in Example A–9:

Example A–9 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–10:

Example A–10 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–11 specifies that all `png` images in the images directory should be rewritten to include the `http://mycdn.com` prefix.

Example A–11 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–12 applies the `http://mycdn.com/` prefix to all images defined by ADF Faces components:

Example A–12 Apply Prefix to a Resource

```
<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. Create rewrite rules to define the replacement URIs for the CDN, using the elements shown in Table A–2.

Note: All attribute values may be EL-bound. However, EL-bound attributes are only evaluated once (at parse time).

Table A–2 CDN URI Rewrite Elements

Element	Definition
`<adf-uri-rewrite-config>`	This `xmlns` value must be set to `http://xmlns.oracle.com/adf/rewrite/config`
`<resource-uris>`	Defines the rules to rewrite the paths to given resources
`<base-resource-uri>`	Defines the base URI for the rewritten path. Multiple `<base-resource-uri>` elements may be defined. This element has three attributes: ▪ `uri`: sets URI for the CDN. This will be used to create the full URI for the given resource." ▪ `secure-uri`: sets the prefix when the URI is secure, for example, "`https://`" ▪ `output-context-path`: determines whether or not to remove the application-specific context path in the rewritten URI. Valid values are either `remove` (the default) or `preserve`.
`<match-pattern>`	A regular expression that is tested against rendered resource URIs. If a match is found, the resource URI is prefixed with the URI specified by the `<base-resource-uri>` element. .Multiple `<match-pattern>` elements may be defined for each <base-resource-uri> element. Note that in order to minimize runtime overhead, the results of resource uri rewriting are cached. To prevent excessive caching, `<match-pattern>` regular 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.
`<match-alias>`	Defines an alias that matches one of the ADF Faces-provided resources. The resource may be one of the following: ▪ `af:documents`: matches ADF Faces' HTML resources (for example, `blank.html`) ▪ `af:images`: matches ADF Faces' and Trinidad's image resources ▪ `af:coreScripts`: matches ADF Faces' boot and core JavaScript libraries ▪ `af:skins`: matches skin-generated style sheets Multiple `<match-alias>` elements may be defined for each `<base-resource-uri>` element.

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–13 shows how an application might be configured to use a CDN.

Example A–13 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"
                       output-context-path="remove">
      <match-pattern>^/.*/images/.*\.png$</match-pattern>
      <match-alias>af:documents</match-alias>
      <match-alias>af:coreScripts</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 JavaScript and CDN

The `af:coreScripts` alias can be used to rewrite ADF's "core" JavaScript libraries (that is, the JavaScript libraries that are present on every ADF page) to a CDN. In addition, `<match-pattern>` regular expressions can be used to rewrite arbitrary (for example, application-defined) JavaScript library URIs. However, in cases where JavaScript libraries are introduced into the page dynamically (for example, as a result of partial page rendering), some origin policy restrictions apply. As a result, JavaScript library URIs that have been rewritten to a cross-origin host may fail to load.

You should limit JavaScript library URI rewriting to those libraries covered by `af:coreScripts` and also in cases where the application-provided libraries are known to be included as part of initial page renders (that is, the libraries are never introduced as part of a PPR request).

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 and 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, however the `adf-settings.xml` 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–14, using the correct settings for your web application root.

Example A–14 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–15 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–15 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 JARs, 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 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–16 shows the `adf-settings.xml` file for the ADF Faces JAR.

Example A–16 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.

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–17 shows a starter `trinidad-config.xml` file.

Example A–17 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>blafplus-rich</skin-family>

</trinidad-config>
```

By default, JDeveloper configures the `blafplus-rich` skin family for a JSF application that uses ADF Faces. You can change this to `blafplus-medium`, `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 20, "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 Application Navigator, double-click **trinidad-config.xml** to open the file in the XML editor.
2. If you are familiar with the element names, enter them in the editor. Otherwise use the Structure window to help you insert them.
3. In the Structure window:
   a. Right-click an element to choose from the **Insert before** or **Insert after** menu, and click the element you wish to insert.
   b. Double-click the newly inserted element in the Structure window to open it in the Property Inspector. 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' : 'blafplus-rich'}`) 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 20, "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`.

> **Note:** Enabling animation will have an impact on performance. For more information, see the "Oracle Application Development Framework Performance Tuning" section in the *Oracle Fusion Middleware Performance Guide*.

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 `blafplus-rich` skin family for a JSF application that uses ADF Faces. You can change the `<skin-family>` value to `blafplus-medium`, `simple`, or to a custom skin definition. For information about creating and using custom skins, see Chapter 20, "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 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.

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 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.6.2.12 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.7 Configuration in trinidad-skins.xml

By default, JDeveloper uses the `blafplus-rich` 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 20, "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 Javadoc 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 Using JavaScript Library Partitioning

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. For more information about JavaScript partitioning, see Section 1.2.1.2, "JavaScript Library Partitioning."

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 about enabling or disabling JavaScript partitioning, see Section A.2.3.15, "JavaScript Partitioning."

A.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.

To create a JavaScript feature:

1. If not already created, create a `META-INF` directory for your component.
2. Right-click the `META-INF` directory, and choose **New** from the context menu.
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. Enter `adf-js-features.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–5.

Example A–18 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>
```

6. 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 A–19 shows the `feature` element for a fictitious custom component that uses popup components (and therefore has a dependency to the popup feature).

Example A–19 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>
```

A.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 Example A–22). 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.

To create JavaScript partitions:

1. Right-click the `WEB-INF` directory, and choose **New** from the context menu.
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. 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 A–5.

Example A–20 XML for adf-js-partitions.xml File

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

</adf-js-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 A–21 shows the `partition` element for the `tree` partition that contains the `AdfRichTree` and `AdfRichTreeTable` features.

Example A–21 JavaScript Partition Configuration

```
<partition>
  <partition-name>tree</partition-name>
```

```
  <feature>AdfUITree</feature>
  <feature>AdfUITreeTable</feature>
  <feature>AdfRichTree</feature>
  <feature>AdfRichTreeTable</feature>
</partition>
```

A.9.3 What You May Need to Know About 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 A.9.2, "How to Create JavaScript Partitions." Example A–22 shows the default ADF Faces `adf-js-partitions.xml` file.

Example A–22 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>
```

A.9.4 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.

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, "Introduction to 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 Introduction to 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.

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)

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

This section gives 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:

```
setMessageDetailConvertBoth(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 date-time example, based on the `dateStyle` and `timeStyle` set in the converter

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 `date`

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 set

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 `invalidDaysOfWeek` is set

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 month

Validate date restriction - Invalid month: The input value is invalid when `invalidMonths` is set

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 weekday

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:

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 BLAF Plus 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 (in Windows) 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.

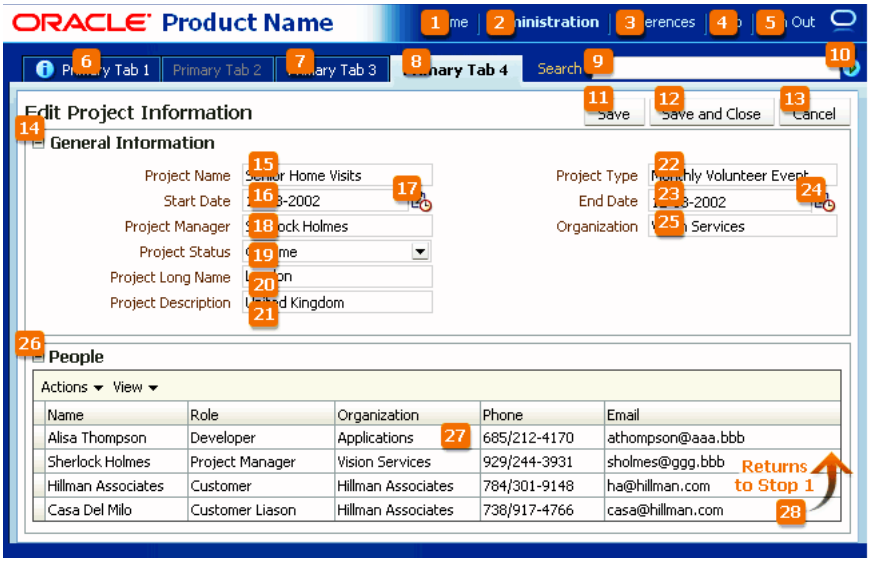
All active elements of the page are accessible by Tab traversal, that is, by using the Tab key to move to the next control and Shift+Tab to move to the previous control. In most cases, when a control has focus, the action can then be initiated by pressing Enter.

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.

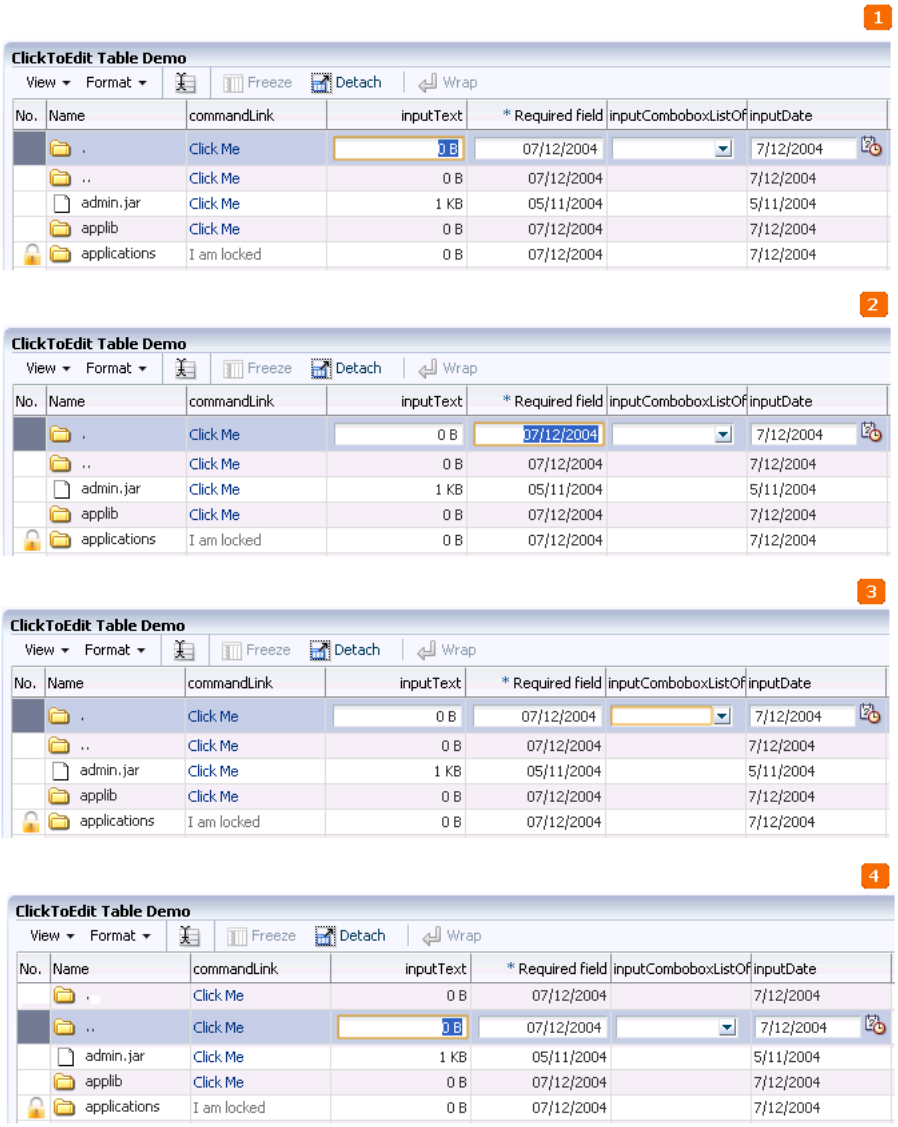
To improve keyboard navigation efficiency for users, you should include a `skip navigation` link at the top of the page, which should navigate directly to the first content-related tab stop.

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 several times 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.

   Pressing the Enter key 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.

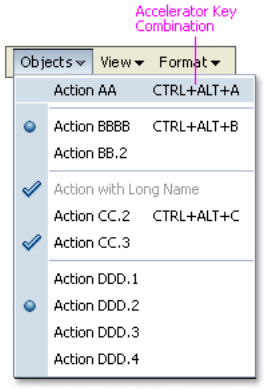
C.3 Accelerator Keys

Accelerator keys bypass menu and page navigation and perform actions directly. Accelerator keys are sometimes 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 metakey and an execution key. The metakey 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 metakey.

BLAF Plus components have some built-in accelerator keys. Custom accelerator keys are supported only in menus, as shown in Figure C–3.

Figure C–3 Accelerator Keys in a Menu



When defining accelerator keys, you must follow these guidelines:

- Accelerator keys must always have alternative interactions, such as direct manipulation with the mouse (for example, clicking a button, or dragging and dropping).
- 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 BLAF Plus-supported browsers (see the browser and system requirements for supported operating systems and browsers in BLAF Plus), 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 metakey may be Ctrl, Ctrl+Shift, or Ctrl+Alt. Ctrl+Alt is the recommended 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 global to the entire page. If a page were to have different menus 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 BLAF Plus 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 BLAF Plus 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+R	Active Data	Check for updated data
Ctrl+Alt+W	Menu Messaging Secondary Windows	Toggle focus between popup and primary window
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–2 in menus when the related component also appears on the same page.

Table C–2 Accelerator Keys Assigned to Optional Components

Accelerator Key	Component	Function
Ctrl+Alt++	Rich Text Editor	Superscript
Ctrl+Alt+-	Rich Text Editor	Subscript
Ctrl+/	Hierarchy Viewer	Switch content panel
Ctrl+Alt+0. . . Ctrl+Alt+5	Hierarchy Viewer	Switch diagram layout
Ctrl+5	Rich Text Editor	Strikethrough
Ctrl+A	File Upload Multi-Select Choice List Multi-Select List Box Pivot Table Rich Text Editor Spin Box Text Box & Area Table	Select all

Table C–2 (Cont.) Accelerator Keys Assigned to Optional Components

Accelerator Key	Component	Function
Ctrl+B	Rich Text Editor	Boldface
Ctrl+Alt+C	Rich Text Editor	Toggle source code editing
Ctrl+E	Rich Text Editor	Center alignment
Ctrl+H	Rich Text Editor	Create hyperlink
Ctrl+Shift+H	Rich Text Editor	Remove hyperlink
Ctrl+I	Rich Text Editor	Italics
Ctrl+J	Rich Text Editor	Full-justified alignment
Ctrl+L	Rich Text Editor	Left alignment
Ctrl+Alt+L	Rich Text Editor	Numbered list
Ctrl+M	Rich Text Editor	Increase indentation
Ctrl+Shift+M	Rich Text Editor	Decrease indentation
Ctrl+Alt+M	Gantt Pivot Table Table Tree Tree Table	Launch context menu
Ctrl+R	Rich Text Editor	Right alignment
Ctrl+Alt+R	Rich Text Editor	Toggle rich text editing
Ctrl+Shift+S	Rich Text Editor	Clear text styles
Ctrl+U	Rich Text Editor	Underline
Ctrl+Y	Rich Text Editor	Redo
Ctrl+Z	Rich Text Editor	Undo
Ctrl+Shift+^	Hierarchy Viewer Tree Tree Table	Go up one level
Esc	Table	Reverse all edits of the row and disable edit mode
Enter Shift+Enter	Table	Navigate to the next or previous cell of the column
Arrow Up Arrow Down	Table Tree Table	Move focus.

C.4 Accelerator Keys for ADF Data Visualization Components

ADF Data Visualization components provide graphical and tabular capabilities for displaying and analyzing data. Table C–3 lists the accelerator keys assigned to ADF Data Visualization components: Gantt chart components, hierarchy viewer components, pivot table components, and geographic map components. For more information about ADF Data Visualization components, see Chapter 23, "Introduction to ADF Data Visualization Components."

Table C–3 Accelerator Keys Assigned to ADF Data Visualization Components

Accelerator 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	Moves the focus. If the focus is on the chart region of the scheduling Gantt component, the arrow key navigation selects the previous or next taskbar of the current row. If the focus is on the time bucket of the resource utilization Gantt component, 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 the Home or End key to pan by a large increment. If the focus is on the node component of the ADF hierarchy viewer component, press Ctrl+Arrow keys to move the focus left or right without selecting the component.
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	Moves the 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 the scheduling Gantt component, 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 the resource utilization Gantt component, 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 the ADF hierarchy viewer, press the Ctrl+Arrow keys to move the focus up or down without selecting the component.

Table C–3 (Cont.) Accelerator Keys Assigned to ADF Data Visualization Components

Accelerator Key	Components	Function
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 the ADF hierarchy viewer component, press and hold the 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	Increases the zoom level. If the focus is on the diagram of the ADF hierarchy viewer component, press number keys 1 through 5 to zoom from 10% through 100%. Press 0 to zoom the diagram to fit within available space.
-	ADF geographic map ADF hierarchy viewer - diagram	Decreases the zoom level. If the focus is on the diagram of the ADF hierarchy viewer component, 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	Launches the context menu.
Home	ADF hierarchy viewer - nodes	Moves the focus to first node in the current level.
End	ADF hierarchy viewer - nodes	Moves the focus to last node in the current level.
Ctrl + Home	ADF hierarchy viewer - nodes	Moves the 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	Goes up one level.
Ctrl+/	ADF hierarchy viewer - nodes	Switches the content panel.
Ctrl+Alt+0	ADF hierarchy viewer - diagrams	Centers the active node and zooms the diagram to 100%.
Tab	ADF hierarchy viewer - nodes Pivot table Pivot filter bar	Moves the focus through the elements.

Table C–3 (Cont.) Accelerator Keys Assigned to ADF Data Visualization Components

Accelerator Key	Components	Function
Esc	ADF hierarchy viewer - nodes	Returns the focus to the containing node. If the focus is on the search panel, close the panel. Closes the Detail window, if it appears while hovering over a node.
Spacebar	ADF hierarchy viewer - nodes Pivot table Pivot filter bar	Selects the active node. Press Ctrl+Spacebar to toggle selection of the active node, and to select multiple nodes.
Enter	ADF hierarchy viewer - nodes Pivot table Pivot filter bar	Isolates and selects the active node. Press Shift+Enter to toggle the state of the node.
/	ADF hierarchy viewer - nodes	Toggles control panel state.
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	Switches diagram layout.
Ctrl+Alt+Arrow keys	Pivot table Pivot filter bar	Changes the layout by pivoting a row, column, or filter layer to a new location. Use Ctrl+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. ■ Move or swap the selected header layer to the specified destination.

Some ADF Data Visualization components provide some common functions to the end user through the menu bar, toolbar, context menu, or Task Properties dialog. You may choose to show, hide, or replace these functionalities. If you hide or replace any functionality, you must provide alternate keyboard accessibility to those functions.

C.5 Access Keys

Access keys move the focus to a specific UI element.

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 pressing 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–4 lists access key combinations for button and anchor components in some common browsers.

Table C–4 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:commandToolbarButton` 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, teams must not use access keys that conflict with the top-level menu access keys in BLAF Plus-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, it is still recommended to pick a letter that is easy for users to remember even without scanning the label. That is often 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, it is recommended to 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.6 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 BLAF Plus 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:

- BLAF Plus 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.7 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 1.2.1.1, "Client-Side Components").

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 5–3, " 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
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.
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

Table D–1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
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
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	

Table D–1 (Cont.) Supported Mobile Browser User Gestures

Mouse Interaction	Touch Gesture	Visual State	Example
	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.

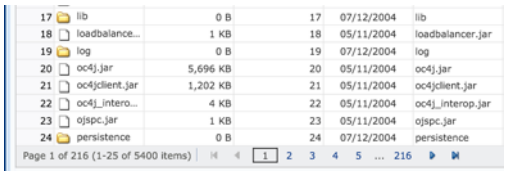
For further optimization, ADF Faces partitions JavaScript, so that the touch device JavaScript is separated from the desktop JavaScript. Only the needed JavaScript is downloaded when a page is rendered. Also, when a touch device is detected, CSS content specific to touch devices is sent to the page. 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`	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 D–2 (Cont.) Component Differences in Mobile Browsers

Component	Functionality	Difference from desktop component
`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	For the root panel component on a page, set the `dimensionsFrom` attribute to `children`, and for all other components that use the `dimensionsFrom` attribute, always set that attribute to `auto`. These settings ensure that the page will flow instead of stretch. For more information, see Section 8.2.1, "Geometry Management and Component Stretching."
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 7.4, "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.

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 Section 20.3.4, "How to Apply Themes to Components."

Figure E–1 Quick Start Layouts With and Without Themes

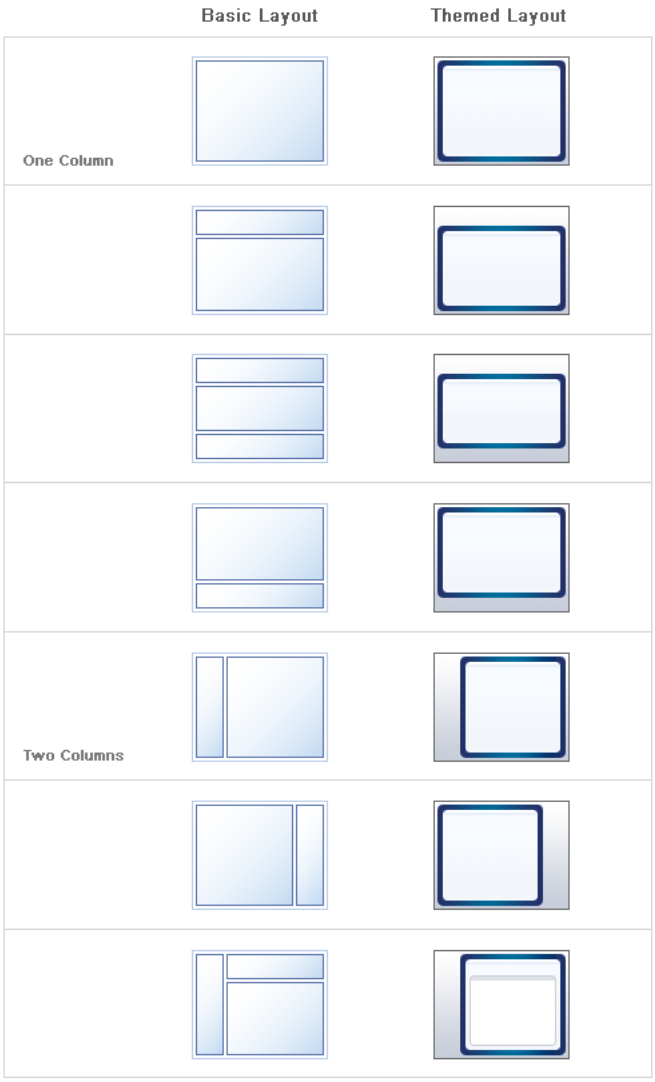
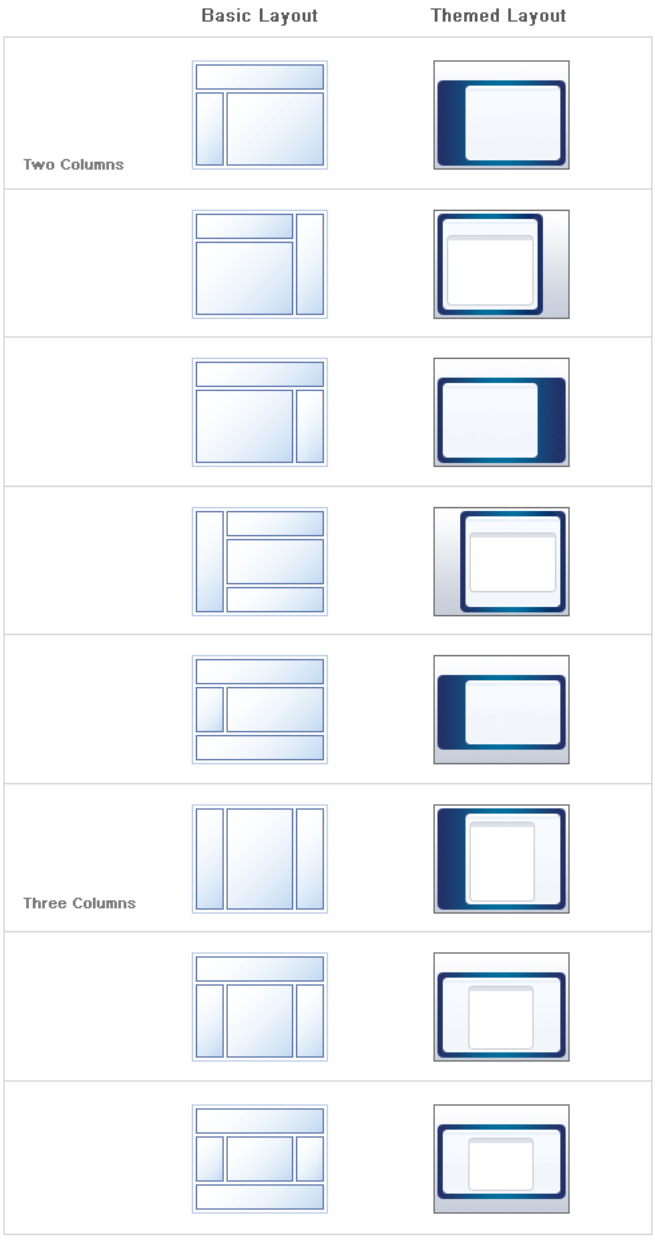


Figure E–2 Quick Start Layouts With and Without Them



F

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 F.1, "Introduction to Troubleshooting ADF Faces"
- Section F.2, "Getting Started with Troubleshooting the View Layer of an ADF Application"
- Section F.3, "Resolving Common Problems"
- Section F.4, "Using My Oracle Support for Additional Troubleshooting Information"

In addition to this chapter, review the *Oracle Fusion Middleware Error Messages Reference* for information about the error messages you may encounter.

F.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 F–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 F–1 Process for Using the Information in this Chapter

Step	Section to Use	Purpose
1	Section F.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 F.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 F.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 F.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`.

F.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 the *Oracle Fusion Middleware Error Messages Reference*. 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 F–2 summarizes the settings that you can follow to configure the view layer of an ADF application for troubleshooting.

Table F–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 F–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.

Optionally, enable performance profiling in the web browser.	Optionally, enable performance profiling in the web browser by setting the following in the `web.xml` file:
	`<context-param>`
	`  <param-name>`
	`  **oracle.adf.view.rich.profiler.ENABLED**`
	`  </param-name>`
	`  <param-value>true</param-value>`
	`</context-param>`
	This is normally handled by all recent version browsers.
	If needed, Oracle recommends setting the log level to INFO level or higher. Output goes to browser console.

F.3 Resolving Common Problems

This section describes common problems and solutions.

F.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 Section 20.2, "Applying Custom Skins to Applications."

F.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 Section 20.7, "Deploying a Custom Skin File in a JAR File."

F.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 Chapter 20, "Customizing the Appearance Using Styles and Skins."

F.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.

F.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
   ```

F.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.

F.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 the *Oracle Fusion Middleware Administrator's Guide*.
2. Look for struck threads, as described in the "Monitor server performance" topic in *Oracle WebLogic Server Administration Console 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.

F.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.

F.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 *Oracle WebLogic Server Administration Console Help*.
2. Check for hung threads.

F.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 F.3.8, "Browser Displays an ADF_Faces-60098 Error."
2. Look for HTTP 404 or 505 errors, as described in Section F.3.9, "Browser Displays an HTTP 404 or 500 Error."

F.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`.